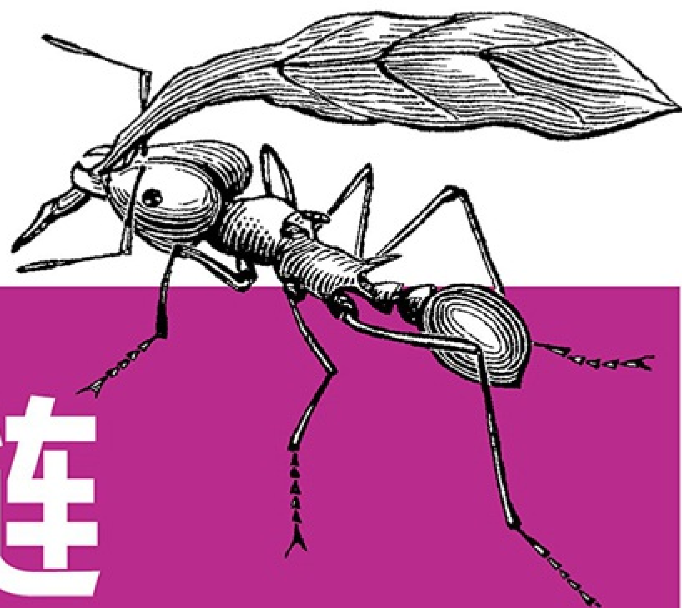
O'REILLY®



和逸金融 HEYI FINTECH　金杜法律研究院



区块链

通往资产数字化之路

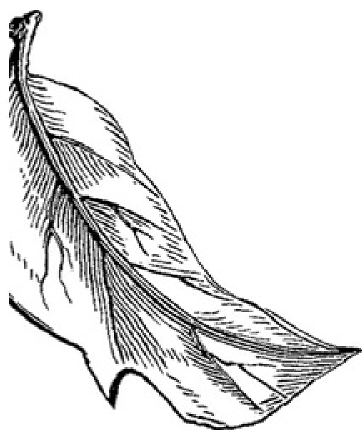
Mastering Bitcoin: Unlocking Digital Cryptocurrencies

肖　风 / 中国万向控股有限公司副董事长

王俊峰 / 金杜律师事务所全球主席

作序推荐





［美］安德烈亚斯·安东诺普洛斯（Andreas M. Antonopoulos）　著

林　华　蔡长春　译　　王　勇　王立荣　译校

中信出版集团

版权信息

书名:区块链：通往资产数字化之路

作者:[美]安德烈亚斯·安东诺普洛斯

译者:林华 蔡长春 王志涵

ISBN:9787508682082

中信出版集团制作发行

当我向普通观众介绍比特币时，我经常被问道：“它到底是怎么工作的？”现在，我对这个问题有了很好的答案，因为任何阅读过《区块链：通往资产数字化之路》的人都将对其工作机制有深入了解，并将具备编写下一代出色的加密货币应用程序的能力。

——加文·安德烈森（Gavin Andresen）比特币基金会首席科学家

比特币和区块链技术正成为下一代互联网的基础组成部分。硅谷最优秀、最聪明的人都在为它努力。安德烈亚斯的书将帮助你进入这场世界金融业的软件革命。

——纳瓦尔·拉维肯特（Naval Ravikant）AngelList联合创始人

《区块链：通往资产数字化之路》是当今比特币研究领域里最好的技术参考书。比特币可能是近十年来最重要的技术。因此，本书绝对是开发者，特别是那些打算基于比特币协议编写应用程序的开发者，所必须拥有的。强烈推荐！

——巴拉吉·斯里尼瓦桑（Balaji S. Srinivasan）（@balajis）

Andreessen Horowitz普通合伙人

比特币、区块链的发明代表了一个全新平台的创立，将使一个与互联网本身一样广泛和多样化的生态系统成为现实。作为一个杰出的思想领袖，安德烈亚斯是写作这样一本书的最佳人选。

——罗杰·冯（Roger Ver）比特币企业家、投资人

中文版序一

从机制设计理论看比特币区块链

交易（Transaction）是人与人之间最基本的经济关系。企业、市场、金融中介、货币体系及与此有关的各种组织，都是重要的便于交易完成的经济制度。在阿罗—德布鲁（Arrow-Debreu）的新古典经济环境中，交易无论是在市场中还是在企业内部进行，结果都完全一样，不同的制度安排仅仅被视为满足帕累托最优（Pareto Optimality）所需的“替代方式”。

但是，近年来，从赫维茨（Hurwicz）、马斯金（Maskin）、梅耶森（Myerson）、梯诺尔（Tirole）、奥斯特罗姆（Ostrom）、威廉姆森（Williamson）等著名学者纷纷获得诺贝尔经济学奖可以看出，理论界和实务界重新认识到不同的制度安排和组织结构在非古典经济环境中对交易费用、激励机制和资源配置效率的重要影响。

事实上，在更接近于现实世界的非古典经济环境中，竞争性的市场机制或政府机构在很多情况下解决不了激励和效率的问题，诸如规则和原则之类的分散决策机制、去中心化机制或许是更为有效的制度。尤其在一些超越国家主权和市场主体的全球公共事务，如气候变化、反恐等问题上，“自主治理、自主组织”具有不可替代的独特作用。

区块链的治理机制也是“自主治理、自主组织”。诞生于2009年的比特币区块链就是这样一个自组织：非营利性组织、产权完全开源、代码随便复制、任何人使用无须得到许可；没有股东会、没有董事会、没有管理层、没有员工、没有资产、没有办公场地、没有资产负债表。

机制设计理论的研究已经有几十年历史了，有关分布式网络、数字货币与智能合约技术的探讨也有着几乎同样长的历史，但它们同时在2008年全球金融危机之后的2009年被世人所认识和追捧（比特币区块链于2009年1月上线，奥斯特罗姆和威廉姆森两位经济学家于2009年10月获得诺贝尔经济学奖）。冥冥之中，我们可以看到这两者之间的紧密联系：2008年金融危机，既使市场无形之手失灵，也使政府有形之手失效！于是大家对“自主治理的自主组织”青睐有加。

比特币区块链可以看作经济学机制设计理论在工程技术层面的创新实践。在对公共资源机制设计有效实践的基础上，区块链技术把它延展到虚拟世界里数字经济的机制设计方面。不管是《失控》（*Out of Control*）的作者凯文·凯利（Kevin Kelly），还是谷歌的首席经济学家哈尔·罗纳德·范里安（Hal Ronald Varian），在论及网络经济学的新规则、新规律时，都或多或少地提到“分布式、去中心、自组织”这个观点。无独有偶，奥斯特罗姆在一项非市场经济制度的研究课题中对美国80个城市警察局进行了研究，结果表明：多中心制度安排的表现要优于大一统的体制。两者在思想上是一脉相承的。比特币区块链9年的实践，也证明了自组织经济治理机制是可行并有效的。

当然，区块链尤其是比特币区块链所谓的“去中心”，其实表述的是一个过程，而不是结果，最终结果不会是完全去中心的。“去中心”的范围也仅限于经济和商业治理方面。就算在经济和商业治理方面，我们看到的也更多是“分中心”“非中心”，而不仅是“去中心”。

多位诺贝尔经济学奖获奖者对“自组织”机制设计的研究也告诉我们：自组织、市场机制和政府管制三者的关系不是颠覆、取代、革命的关系，而是相互补充、相得益彰的关系。经济学家们的研究成果，是区块链这种跨时空、跨主体、全球化、数字化经济现象的理论基础。

林华教授邀我为《比特币：通往资产数字化之路》中文版写序，遂从经济学机制设计理论的角度谈一些对比特币区块链的学习体会，希望

能够为尊敬的读者阅读理解本书提供一点帮助。比特币也许会消失，但比特币区块链在自组织治理机制方面的工程实践意义，必将像明灯一样照亮人类的资产数字化迁徙之路！

肖风

中国万向控股有限公司副董事长

中文版序二

比特币是什么

这几年科技行业最“火”的除了人工智能，估计就是区块链了。数字货币，作为区块链的一种典型应用，首先吸引了大家的眼球。2017年以来，各种数字货币价格暴涨，而一家家区块链初创公司通过首次币发行（Initial Coin Offering，简称ICO），发行各种代币，更是可以在极短的时间内募得几亿美元的资金。除了数字货币（以及其他代币）市场的火爆，区块链在商业市场的应用也是遍地开花。金融机构、科技企业纷纷试点区块链应用，在供应链、慈善捐赠、跨境支付、农业生产等不同领域都能看到区块链的身影。

提起区块链，大家第一个想到的肯定是比特币。不管读者对比特币抱着什么样的看法，毋庸置疑的是，第一个得到广泛认可的数字货币是比特币。区块链的概念正是发源于比特币。研究区块链而不谈及比特币，总是让人感觉少点东西。那么比特币到底是什么？

比特币确切的发布时间，很难说清楚，但是我们可以从几个关键事件大致判断其出现的时间：一个是2008年11月，中本聪（Satoshi Nakamoto）在一个密码学讨论组中发表了一篇论文《比特币：一种点对点的电子现金系统》（*Bitcoin:A Peer-to-Peer Electronic Cash System*）；另外一个是在比特币的创世区块上引用了2009年1月3日《泰晤士报》（*The Times*）上的一句话：“财政大臣正处于实施第二轮银行紧急援助的边缘”（Chancellor on brink of second bailout for banks）。从上面两件事可以判断，比特币的思想在2008年11月前已经成熟，而其发布时间应

该不早于2009年1月3日。

比特币创世区块上引用的那句话，结合当时金融危机的背景及比特币的这个币（coin）的含义，看起来挑衅意味十足，暗示传统金融已经病入膏肓，只有重新设计一种全新的货币体系，才能避免金融危机。

那么中本聪认为的这种理想货币是怎么实现的呢？他在论文的摘要中是这么写的："我们在此提出一种解决方案，使现金系统在点对点的环境下运行，并防止双重支付问题。该网络通过随机散列对全部交易加上时间戳（timestamps），将它们合并入一个不断延伸的基于随机散列的工作量证明（Proof-Of-Work）的链条作为交易记录，除非重新完成全部的工作量证明，否则形成的交易记录将不可更改。最长的链条不仅将作为被观察到的事件序列的证明，而且被看成来自CPU计算能力最大的池。只要大多数的CPU计算能力没有打算合作对全网进行攻击，那么诚实的节点将会生成最长的、超过攻击者的链条。这个系统本身需要的基础设施非常少。信息尽最大努力在全网传播即可，节点可以随时离开和重新加入网络，并将最长的工作量证明链条作为在该节点离线期间发生的交易的证明。"这段话听起来有点拗口，简单来说，比特币是一种数字货币，它建立在点对点的网络之上，基于密码学的方法，大量计算机的分布式计算，经由算力的累积，形成不可篡改的区块链，从而解决双重支付（double-spending）、中心化信任等问题。

当然，近十年过去了，金融行业的基础——货币，并没有被中本聪的这个凭空而出的理想的电子现金系统取代。但随着比特币的面世带来的思想变革和技术革新却实实在在地对现代社会带来了巨大的影响。

首先是分布式的思想。比特币网络没有中心节点，所有节点的身份都是平等的，共同维护网络的安全、稳定运行。网络本身是开放的，节点可以随时选择加入或者退出。

然后是自治的思想。由于没有中心，就需要有一种方法使所有节点

能够自发的形成共识。这本书的封面，画的是一种蚂蚁，成百上千万只的这种蚂蚁组成的组织，可以在简单规则的约束及互相制约影响下，缔造出一个完美的帝国。比特币也是如此，共识算法让无数参与者共同形成了一个完美的生态体系。

更加重要的还有区块链的概念。当然，区块链不能脱离上面提到的思想。基于密码学理论的共识算法，分布式的计算与存储的区块链可以确保数据的不可篡改性，并能做到分布式（或者弱中心化）的信任。这种特性由于与众多应用场景相契合，极具想象空间。有人甚至把区块链形容为互联网之后的人类的最大创新，或者叫价值互联网。

当前国内出版的区块链相关书籍讲应用的多，涉及技术细节的少，偏技术的主要有林华等人之前翻译的普林斯顿的教科书《区块链：技术驱动金融》，主要以比特币为例，介绍了区块链的基本原理，偏向技术普及，深度上尚有所欠缺。这本《区块链：通往资产数字化之路》则专注比特币的讲解，在深度、广度上都远超目前已出版的数字货币相关书籍。

该书的作者安德烈亚斯·安东诺普洛斯（Andreas M.Antonopoulos）是比特币领域的知名专家，他这本书的编排非常有特点，按他自己的说法，是采用讲故事的方式介绍比特币的。全书看下来，确实如此。作者的第一个故事就是介绍如何获取第一个比特币，如何使用比特币购买一杯咖啡，如何确认支付已经完成。一个完全不明白比特币的人看完第一个故事，也能立即对比特币系统有个直观的概念。针对故事的每个细节，作者通过不断深入的方式，逐步揭示了整个比特币系统的运行机制。一本书看下来，相信读者会对比特币了然于胸。如果有编程基础，参照书中的程序示例，也许创建一种自己的数字货币也不是问题。

在我看来，关于比特币，关于区块链，还没有发现有第二本书能像这本书讲得如此透彻而生动的。

值此新书出版之际，特向所有对区块链的爱好者隆重推荐本书！

王俊峰

金杜律师事务所全球主席

译者前言

想加入这场暴风雨般席卷全球金融业的技术革命吗？《区块链：通往资产数字化之路》是你通往看似纷繁复杂的比特币世界的指南，它为你进入这个货币互联网世界，提供了必要的知识。不管你是正在构建下一个杀手级应用，还是在投资一个初创企业，或者只是对技术好奇，这本实用的书都是必不可少的。

比特币，第一个成功的去中心化数字货币，虽然仍处于起步阶段，却已经带来了全球范围内数十亿美元规模的经济效应。这种经济活动对任何有相关知识和热情的人都是开放的。《区块链：通往资产数字化之路》将为你提供必要的知识（不包括热情）。

这本书包括以下方面：

●比特币的概况——非技术用户、投资者、企业高管的理想选择。

●比特币及加密货币的技术基础——为开发人员、工程师及软件和系统架构师量身定制。

●比特币去中心化网络、点对点架构、交易生命周期及安全原则等相关细节的详细介绍。

●基于比特币和区块链的分支的发展，包括替代链、替代币和替代应用程序等。

●通过用户的故事、优雅的类比、示例、代码片段等阐述关键的技术和概念。

原版前言

为什么撰写本书

我第一次偶遇比特币是在2011年年中，当时的第一反应就是："书呆子货币！"然后就把它扔到一边长达6个月，这让我白白失去了一次了解它的机会，更不用说深入认识它的重要性了。其实，我熟悉的很多聪明人面对比特币时也是跟我一样的反应，这倒是给了我一些安慰。第二次接触比特币，是在一个邮件列表讨论群，这次我决定好好拜读一下中本聪发表的白皮书，研究一下权威来源是怎么说的，看看比特币到底是什么。我依然记得，当读完这9页文字后，我有多震撼！比特币不仅仅是一个数字货币，它的意义已远超数字货币本身，它是构成信用网络的基础，可以在更广泛的领域应用。意识到"这不是一种货币，而是一个去中心化的信用网络"后，我开始了4个月的比特币之旅，我饥渴地吸收所有能收集到的有关比特币的信息，我变得如此痴迷，以致每天都要花12个小时甚至更多的时间盯着屏幕，阅读、写作、编码，希望能够学到有关的一切。由于饮食不正常，我瘦了18斤，我决定从这种状态中摆脱出来，专心投入比特币的研究工作。

两年来，在创建了几家小型创业公司以探索各种比特币相关的服务和产品后，我觉得是时候开始写我的第一本书了。比特币使我有了疯狂的创意，充实了思想；这是自互联网以来我见过的最激动人心的技术。我要把对这个伟大技术的热情分享给更多朋友。

目标受众

本书主要面向开发人员。如果你已经掌握一门编程语言，这本书可以告诉你加密货币是如何工作的，怎么使用它们，怎么开发基于它们的应用。前面几章同样也适合希望深入了解比特币和加密货币内部运行机制的非开发人员，这几章将对这些方面进行了深度介绍。

关于封面上的昆虫

切叶蚁是一种在超个体群落中展现出极度复杂性行为的生物，但是落实到个体上，单个蚂蚁只是在一套简单的规则驱动下与外部交互，交换化学味素（信息素）。在维基百科中是这么描述的："除了人类，切叶蚁是地球上最大、最复杂的动物社会。"切叶蚁实际上并不吃树叶，而是利用树叶来培育真菌，这些真菌是切叶蚁群体的主要食物来源。看到了吗？这些蚂蚁实际上是在进行农耕！

虽然蚂蚁是一种"种姓基础"的社会，有一个蚁后专门负责繁殖后代，但是它们并没有一个中央政权，或者说在它们的群落中并没有领导者。上百万只成员组成的蚁群所展示出的高度智能化及其复杂的行为，仅仅只是个体在与社会网络交互中产生的一种自然属性。

自然界证明了去中心化的系统也同样可以具有弹性，可以在没有中央集权、没有层级、没有复杂结构的社会中产生不可思议的复杂性。

比特币是一个高度复杂的去中心化信用网络，它能够支持各种各样的金融流程。但是，即使在这么复杂的情况下，网络中的单个节点也仅仅是遵循几个简单的数学规则。多个节点间的交互导致复杂行为的产生，个体节点自身并不需要拥有复杂性或者信用机制。就像一个蚁群，比特币网络是由无数遵循简单规则的节点一起组成的，不需要中心协作却能完成各种不可思议事情的弹性网络。

本书的约定

本书遵循以下排版约定：

粗体

代表新名词、URLs（统一资源定位符）、email地址、文件名，以及文件扩展名。

固定宽度

用作程序列表，段落中对程序元素的引用，比如变量、函数名、数据库、数据类型、环境变量、声明、关键字等。

黑体固定宽度

表示命令或者其他需要用户准确输入的文字。

斜体固定宽度

表示需要被用户提供的值所替代或者通过上下文确定的值所替代。

这个图标表示一个提示、建议或者一般注解。

这个图标表示警告或者注意。

这个图标表示对正文内容的补充解释。

代码示例

代码示例通过Python、C++编程语言提供，并且使用类Unix系统（如Linux、Mac OS X）的命令行执行。所有代码片段在GitHub代码库

均能找到，位于主库（**https://github.com/aantonop/bitcoinbook**）的**code**子目录下。你可以分叉（fork）一个本书的代码库，并且尝试运行它们，或者也可以通过GitHub提交勘误。

所有代码均可以在大部分安装了相应语言编译器或者解释器的操作系统上运行。如果需要，本书也提供了基础安装指南以及每一步输出的范例。

为了打印美观，一些代码段以及代码的输出进行了重新格式化，在这种情况下，长的代码行通过“\\”分割，并紧跟换行符。当重新编译或执行这些范例时，需要把这两个字符删除，并把代码行进行合并，这样就能看到与文中一样的输出结果。

所有代码段均尽可能采用真实的数值进行计算，所以你可以自己编译执行这些代码并且看到同样的结果。举例来说，私钥和对应的公钥以及地址均为真实的值。示例中的交易、区块、区块链的索引也都是从比特币区块链的公共账本中获取的，所以你可以在任意比特币的系统上直接查看。

使用代码范例

这本书是为了帮助你完成工作的，通常来说，如果本书中提供了示例代码，都可以直接在你的程序和文档中使用，而不需要联系我们获得许可，除非你复制本书的大段代码。比如，在程序中引用书中的几段代码，你并不需要得到我们的许可；但是如果你出售或者通过CD-ROM传播从O’Reilly的书中获得的代码，你确实需要得到授权；通过引用本书以及书中代码来回答一个问题并不需要得到授权，但是引用大段代码到你产品的文档中则需要授权。我们感谢但不强求你的引用说明。一个参考文献说明通常包含标题、作者、出版商，以及书的ISBN。比

如：“*Mastering Bitcoin* by Andreas M.Antonopoulos（O’Reilly）.Copyright 2015 Andreas M.Antonopoulos,978-1- 449-37404- 4.”

本书的某些版本是以开源版权的方式提供的，比如CC-BY-NC（creativecommons.org），这种情况下版权协议遵照相应的开源版权说明。

如果你觉得在使用代码的过程中，可能已超过了合理使用范围或者以上提到的授权说明，请通过**permissions@oreilly.com**联系我们。

Safari®在线图书

Safari Safari在线图书（Safari Books Online）是一个按需提供的数字图书馆，它同时以书本和视频的方式提供技术及商业领域世界顶级作者的专业内容。

技术专家、软件开发人员、web设计师、商业和创意专家将Safari在线图书作为他们研究、解决问题、学习和认证培训的主要信息来源。

Safari在线图书为机构、政府机关和个人提供了一定范围的产品组合和定价计划，订阅用户可以访问来自O’Reilly Media,Prentice Hall Professional, Addison-Wesley Professional,Microsoft Press，Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology等出版商的数据库，可以全文搜索书籍、培训视频和正式出版前的书稿。欢迎访问我们的网站以获取更多相关信息。

如何联系我们

请将有关本书的意见和问题通过以下地址发送给出版商：

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938（美国或加拿大）

707-829-0515（国际或本地）

707-829-0104（传真）

我们也有一个关于本书的网站，在那儿可以看到书籍的勘正、例子及其他一切相关的信息，网页地址是**http://bit.ly/mastering_bitcoin**。

反馈意见或者询问本书相关的技术问题，请发邮件至**bookquestions@oreilly.com**。

获取更多我们的书籍、教程、会议、新闻信息，请访问网站：**http://www.oreilly.com**。

我们的脸书（Facebook）地址是**http://facebook.com/oreilly**。

在推特（Twitter）上关注我们**http://twitter.com/oreillymedia**。

在YouTube上观看我们的视频**http://www.youtube.com/oreillymedia**。

致谢

这本书的出版凝聚了很多人的努力和贡献。在此，我要感谢所有帮过我的朋友、同事甚至很多陌生人，这些人与我一起努力完成了这本关于加密货币和比特币的纯技术书籍。

实际上，要完全将比特币技术和比特币社区区分开是不可能的，这本书不仅是这个社区的产品，也是关于这个技术的书籍。在撰写这本书的过程中，自始至终，我不断被整个比特币社区鼓舞，得到很多支持和激励。最重要的，这本书使我这两年来真正成为这个伟大社区的一员，我无法用语言来表达对你们接受我进入这个社区的感激之情。太多的人，我已经无法一一提及名字——在正式会议、社区活动、小型研讨会、大型聚会、比萨聚餐、小型私人聚会等面对面场合，以及在Twitter、reddit、bitcointalk.org、GitHub等在线交流中，大家为本书的撰写提供了各种帮助，并施加了各种影响。你能看到的书中每一个主意、推导、问题、答案，均在一定程度上在我与社区的互动中得到了测试或者改进。感谢所有人的支持，没有你们，这本书是不可能出现的，我会永远感激你们！

当然，我成长为一个作家的历程在写第一本书之前很久就开始了。我的母语（以及教育经历）是希腊语，所以我在大学的第一年不得不补修英语写作。我要感谢我的英语写作老师戴安娜·科达斯（Diana Kordas），她帮助我树立了信心并且掌握了英语写作的技巧。后来，作为一个技术专家，我通过给《网络世界》（*Network World*）撰写有关数据中心的议题进一步提高了写作技巧。我要感谢约翰·迪克斯（John Dix）和约翰·格兰特（John Gallant），他们给了我作为《网络世界》专栏作家的第一份写作工作，我的编辑迈克尔·库尼（Michael Cooney）及我的同事乔娜·迪尔·约翰逊（Johna Till Johnson），他们帮忙修改我的专栏文章，使之适合出版。持续4年、每周500字的写作经历给了我足够的经验，并最终考虑成为一个作家。感谢简·德·维拉（Jean de Vera），

是她最早鼓励我成为一个作家，并且总是相信并坚信我能完成一本属于我自己的书。

同样感谢那些O’Reilly的同事，当我提交我的书稿时，他们向我提供参考资料，审阅书稿，给了我很大的支持。特别需要感谢的是：约翰·格兰特，格里高利·奈斯（Gregory Ness），理查德·斯迪侬（Richard Stiennon），乔尔·斯尼德（Joel Snyder），阿旦姆·B.勒文（Adam B. Levine），桑德拉·迪特隆（Sandra Gittlen），约翰·迪克斯，乔娜·迪尔·约翰逊，罗杰·维（Roger Ver），乔·马托尼斯（Jon Matonis）。特别感谢理查德·卡甘（Richard Kagan）和迪蒙·马托斯科（Tymon Mattoszko），他们对书稿的早期版本进行了审阅，马修·欧文·泰勒（Matthew Owain Taylor）对书稿进行了编辑。

感谢克里克特·刘（Cricket Liu），O’Reilly出版的《**DNS**和**BIND**》（*DNS and BIND*）的作者，他将我介绍给了O’Reilly。同样感谢O’Reilly的迈克尔·鲁基德斯（Michael Loukides）和阿利森·麦克唐纳（Allyson MacDonald），他们经过几个月的辛勤努力才使本书的出版成为可能。当我由于生活干扰而错过截止日期、推迟交稿时，阿利森给予了足够的耐心。

开始几稿的前面几章特别困难，因为比特币实在是一个难以阐明的主题。每次我想抽取比特币技术的一个主题时，总是不得不把整件事情一起扯进来。挣扎着想将一个复杂的主题尽量用易于理解的语言进行阐述时，我总是不停地“卡壳”，非常沮丧。最终，我决定通过讲述人们使用比特币的故事来组织整本书的思路，这使撰写过程变得简单了许多。我必须感谢我的朋友和导师理查德·卡甘，他帮助我阐明了故事，度过了文思枯竭的阶段；帕姆拉·摩根（Pamela Morgan）帮我审阅了各章的早期版本，并且解决了一系列棘手的问题，使文章变得更好。同样，感谢旧金山的比特币开发小组和塔里克·路易斯（Taariq Lewis）——该小组的联合创始人，是他们帮忙测试了早期的材料。

在撰写本书的过程中，我把早期的书稿放在了GitHub上，并邀请了大家提意见。在此过程中，我收集到了超过100条的评论、建议、修正，这些贡献均已在《早期版本（GitHub反馈意见）》中进行了公开致谢。特别感谢敏·T.阮（Minh T. Nguyen），他自愿整理了GitHub上的反馈意见，并增加了很多他自己的意见。也感谢安德鲁·诺格勒（Andrew Naugler）提供了信息图表的设计。

书成稿后，经过了几轮的技术审阅。感谢克里克特·刘和罗恩·兰特兹（Lorne Lantz）对文稿的周密审阅、评论和支持。

还有几位比特币开发者贡献了代码示例、审阅、评论和鼓励。感谢阿米·塔基（Amir Taaki）和埃里克·沃斯奎尔（Eric Voskuil），他们提供了代码示例和很多很好的评论；感谢维塔利科·布特林（Vitalik Buterin）和理查德·基斯（Richard Kiss），他们提供了椭圆曲线算法方面的帮助，并贡献了部分代码；感谢盖文·安德森（Gavin Andresen），他提供了勘正、评论和鼓励；感谢米查里斯·卡嘎基斯（Michalis Kargakis），他提供了评论、建议和btcd的编写。

我想把我的爱和本书献给我的母亲特丽莎（Theresa），她在一所墙上摆满书的房子里把我抚养成人。她在1982年就给我买了第一台电脑，虽然她自认为是个技术恐惧症患者。我的父亲门内劳斯（Menelaos），一个土木工程师，刚刚在他80岁高龄的时候出版了第一本书，是他教会我逻辑和分析思维，并使我爱上了科学和工程。

感谢所有支持我经历了这段历程的朋友们！

早期版本（GitHub反馈意见）

非常多的反馈者在GitHub上对早期版本提供了很好的评论、纠正，以及其他帮助。感谢你们对本书提出的所有宝贵意见，以下是部分反馈

意见者的列表，括号中是他们的GitHub ID：

●Minh T. Nguyen，《早期版本（GitHub反馈意见）》编辑（enderminh）。

●Ed Eykholt（edeykholt）。

●Michalis Kargakis（kargakis）。

●Erik Wahlström（erikwam）。

●Richard Kiss（richardkiss）。

●Eric Winchell（winchell）。

●Sergej Kotliar（ziggamon）。

●Nagaraj Hubli（nagarajhubli）。

●ethers。

●Alex Waters（alexwaters）。

●Mihail Russu（MihailRussu）。

●Ish Ot Jr.（ishotjr）。

●James Addison（jayaddison）。

●Nekomata（nekomata-3）。

●Simon de la Rouviere（simondlr）。

●Chapman Shoop（belovachap）。

●Holger Schinzel（schinzelh）。

●effectsToCause（vericoin）。

●Stephan Oeste（Emzy）。

●Joe Bauers（joebauers）。

●Jason Bisterfeldt（jbisterfeldt）。

●Ed Leafe（EdLeafe）。

术语表

这张术语表包含了很多与比特币相关的名词，这些名词的使用将贯穿本书，所以请标记此页以便快速查阅。

地址（address）

比特币地址看起来就像这样：1DSrfJdB2AnWaFNgSbv3MZC2m74996JafV，由一串以1开头（数字1）的字母和数字组成。就像你让别人往你的电子邮箱地址发送邮件一样，你也可以让别人往你的比特币地址发送比特币。

比特币改进提案（BIP）

比特币改进提案是比特币社区成员提交的一系列用以改进比特币的提案。比如，BIP0021就是一个改进比特币统一资源标识符（Uniform Resource Identifier，简称URI）的提案。

比特币（bitcoin）

货币单位、网络，以及软件的名称。

区块（block）

一组交易的集合，标上了时间戳，并包含前个区块的指纹。区块头经过哈希计算生成工作量证明，从而验证所有交易的有效性。经过验证的区块将通过网络共识添加到主区块链中。

区块链（blockchain）

有效区块的列表，每个区块均指向其前序区块，直到创世区块（genesis block）。

确认（confirmations）

一旦一个交易被包含到区块中，它就有了一个确认。当同一条区块链上的另一个区块被挖矿发现后，这个交易就有了两个确认，以此类推。通常认为，六个或者更多的确认已足够证明这笔交易无法撤销。

难度（difficulty）

一个全网设定，用于控制全网需要投入多少计算能力来生成一个工作量证明。

难度目标（difficulty target）

一个难度值，使得全网算力平均10分钟左右找到一个新的区块。

难度目标重估（difficulty retargeting）

每挖出2106个区块后，全网基于前面2106个区块的哈希算力重新计算难度值。

费用（fees）

交易发送方通常会在提交到网络的交易中包含一定的交易费用。绝大部分交易要求至少0.5毫比特币的交易费用。

哈希（hash）

二进制输入的一种数字指纹。

创世区块

区块链的第一个区块，用于初始化加密货币系统。

矿工（miner）

一种通过不断重复计算哈希找到有效区块工作量证明的网络节点。

网络（network）

一种点对点网络，广播所有交易和区块信息到网络上的所有比特币节点。

工作量证明（Proof-Of-Work）

一段需要经过大量计算才能获得的数据，矿工们需要找到一个基于SHA256算法的数字解决方案以达到全网的难度要求。

奖励（reward）

一笔包含在每个新区块中的、作为找到工作量证明的矿工奖励的资金。目前每个区块的奖励是25个比特币[注]。

密钥（secret key）

或称为私钥（private key），一个用于解锁相应接收地址的比特币

的保密数字，一个密钥看起来像这样：5J76sF8L5jTtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3ibVPxh。

交易（transaction）

简单来说，就是比特币从一个地址到另一个地址的转移。更确切地说，一个交易是经过签名的，代表价值传递的数据结构。交易通过比特币网络传递，被矿工收集并打包进区块中，使其永久保存在区块链上。

钱包（wallet）

保管你的所有比特币地址和密钥的软件。你可以通过它来发送、接收和保管比特币。

---

1. 此为作者写书时的奖励金额，翻译此书时是12.5比特币。——译者注

第1章 欢迎来到比特币世界

什么是比特币

比特币是一系列构成数字货币生态系统的概念和技术的组合。比特币的货币单位也叫作“比特币”，用于存储和传递价值。比特币用户间的通信主要通过比特币协议在互联网上进行，也可以在其他通信网络中进行。比特币协议栈是开源的，可以在各种不同的计算设备上运行，包括笔记本电脑、智能手机等，用户接入比特币网络非常方便。

用户可以在网络中传递比特币，完成一切传统货币可以完成的事情，包括买卖商品、转账给特定的个人或者组织、发放贷款等。比特币可以进行买卖，也可以在专业的货币交易所中与其他货币进行兑换。比特币交易快速、安全，并且没有边界，从某种程度上来说，是互联网上的一种完美的货币形式。

不像传统货币，比特币是一种彻底的虚拟货币，没有物理货币，甚至连电子货币本身都不存在。货币隐含在发送者与接收者进行价值交换的交易当中。比特币用户拥有自己的密钥，用以证明比特币网络中的交易所有权，并实现交易消费，或将其传递给新的接收者。这些密钥通常存储于用户计算机的数字钱包中。拥有解锁交易的密钥是花费比特币的唯一要求，这也就把对比特币的控制权完完全全地交给了用户。

比特币是一个分布式、点对点系统。网络中没有“中央”服务器，也没有控制点。比特币是由被称为“挖矿”的过程产生的，它是一种在验证比特币交易的过程中竞争解决一类数学问题的机制。任何比特币网络的参与者（运行完整的比特币协议栈的人）都可以成为挖矿者，他们可以使用自己的计算机的处理能力去验证和记录交易。平均每隔10分钟，总有人能完成过去10分钟所产生的交易的验证过程，并因此获得全新产生的比特币的奖励。本质上说，比特币的挖矿机制使中央银行的货币发行

和清算机制得以去中心化，中央银行的功能被这种全局竞争机制替代了。

比特币协议内建的算法规范了全网挖矿的行为。矿工记录处理交易区块的难度可以动态调整，确保了不管网络上有多少矿工（CPU）同时在工作，最终都能维持大致10分钟挖到一个区块的速度。协议同样规定了每隔4年，新比特币的创建速度将减半，这将比特币的总量限制在了2100万的总量内。因此，比特币的发行量与预测的曲线可以尽量靠近，直到2140年达到2100万的总量。鉴于比特币发行量递减，长期内比特币是维持通货紧缩的。此外，比特币系统的限制使得无法通过“印钞”导致通货膨胀。

除了表现出来的货币属性，比特币也是协议的名称，它是一个网络，一种分布式计算的创新。比特币作为货币仅仅是基于这个发明的第一个真正的应用。作为一个开发者，我认为比特币类似于货币的互联网，它是一种传播价值并通过分布式计算保护数字资产所有权的网络。比特币能做的要比你第一眼看到的多得多。

在本章中，我们将从解释主要的概念和名词入手，下载所需软件，并使用比特币进行简单交易。接下来的几章中，我们将一步步解释使比特币成为可能的技术细节，并深入了解比特币网络及其协议的运行机制。

比特币之前的数字货币

数字货币的出现与密码学的发展紧密相关。考虑到利用数字来表示货物或者服务的价值所面临的根本挑战，数字货币的出现也就不足为奇了。任何接受数字货币的人都要面对两个根本问题：

1. 我能相信钱是真实的而不是伪造的吗？

2. 我能确定没人会声明钱是他的而不是我的吗（又被称为“双重支付”问题）？

纸币的发行者为了防止假钞，采用愈加复杂的印钞纸和更为先进的印刷技术来印制钞票。使用物理货币的情况下，解决重复支付的问题非常简单，因为同一张钞票不可能同时在两个地方出现。当然，传统货币也经常以电子的方式进行存储和传递，在这种情况下，防伪和防止双重支付是通过中央机构对电子交易进行集中清算实现的，这个机构拥有货币流通的全局视角。对于数字货币而言，它无法依靠防伪油墨、全息安全线来保障安全，而密码学提供了基础的保证，从而实现对用户合法价值的信用。特别地，加密数字签名算法使用户可以对数字资产或数字资产的交易进行签名。利用合理的架构，数字签名也能用于解决双重支付的问题。

20世纪80年代，密码学越来越为人所熟知，并得到了越来越广泛的应用，许多研究人员开始尝试使用密码学创建数字货币系统。这些早期的数字货币项目所发行的货币通常由国家法定货币或者贵金属进行“背书”。

虽然这些数字货币系统也能运行，但是它们是中心化的，很容易被政府或者黑客攻击。早期数字货币与传统银行系统一样，利用中央清算机构定时处理所有交易。不幸的是，这些数字货币系统大都成为政府担忧的目标，最终因诉讼失败而消失了。也有因为母公司突然破产清算而悲壮倒闭的。为了应对反对者（不管是合法政府还是罪恶因素）的干扰，防止单点攻击，都有必要引入一个去中心化的数字货币系统。比特币就是这样一个系统，当它被设计出来时就是完全去中心化的，不需要任何中央集权机构，也不需要可被攻击且容易崩溃的单一控制节点。

比特币是几十年来密码学和分布式系统研究的巅峰之作，它汇集了4个方面的创新，形成了一个单一的强大组合。比特币系统由以下部分组成。

●去中心化的点对点网络（比特币协议）。

●公共交易账本（区块链）。

●去中心化的基于数学的确定性的货币发行体系（分布式挖矿）。

●去中心化的交易验证系统（交易脚本）。

比特币的历史

2008年，一个化名为中本聪（Satoshi Nakamoto）的人公开发表了一篇叫作《比特币：一个点对点数字货币系统》（*Bitcoin:A Peer-to-Peer Electronic Cash System*）的论文，比特币从此出现在世人面前。中本聪结合之前发明的几种数字货币，如b-money、HashCash等，创建了一个完全去中心化的货币系统，它不依赖于任何中央机构进行货币发行或者交易结算、验证。其最主要的创新在于利用分布式计算系统（被称为“工作量证明”算法）来组织10分钟一次的全局“选举”，使去中心化的网络形成对交易状态的共识。这个机制优雅地解决了双重支付问题，避免了货币能被多次消费的问题。之前，双重支付一直都是数字货币系统的弱点，以致不得不引入一个中央清算机构来完成交易清算。

比特币网络开始于2009年，它由基于中本聪发布的并被大量其他程序员修订过的核心客户端发展而来的。比特币发明以来，为比特币提供安全性和弹性保障的分布式计算已经实现了指数级的增长，现在其计算能力已超过了全世界最强大的超级计算机的处理能力。基于比特币与美元的汇率估算，比特币的全部市场容量介于50亿到100亿美元之间。比特币网络迄今处理的最大一笔交易额为1.5亿美元，瞬间就完成了传递和处理，并且没有产生任何费用。

中本聪自2011年4月起从公众视野中消失，将开发代码和建设网络的责任交给了一个活跃的志愿者小组。这个比特币背后的人（或者群体）的身份依然未知。但是，不管是中本聪还是任何其他人均无法对比特币系统进行控制，这个系统只依赖于完全透明的数学法则。发明本身是开创性的，并且已经在分布式计算、经济学、计量经济学等领域中产生了新的学科。

分布式计算问题的一个解决方案

中本聪的发明也是对之前未能解决的分布式计算问题（“拜占庭将军问题”）的一个实用解决方案。简单来说，问题在于如何在一个不可靠且存在潜在背叛风险的网络中交换信息并达成共识。中本聪的解决方案是在一个没有中央可信节点的情况下，利用工作量证明来达成共识，它标志着分布式计算科学的一个重大创新，它的适用性要远远超越货币领域。比特币可以在去中心化的网络中达成共识，从而证明选举、彩票、资产注册、数字公证等活动的公正性。

比特币的使用、用户，以及他们的故事

比特币是一种技术，但它所代表的货币，本质上是人与人之间实现价值交换的语言。我们来看看使用比特币的人们，并通过他们的故事来了解几种常见的货币和协议使用场景。我们还将在本书中不断重复使用这些故事，以展示数字货币在现实生活中的应用，以及比特币中的各种技术是如何使这些应用成为可能的。

北美低价值零售

爱丽丝（Alice）住在北加利福尼亚州（简称加州）湾区，她从她的技术迷朋友口中听说了比特币，也想开始使用它。我们将一路追随她的故事，从她了解比特币开始，然后从朋友处获得一些比特币，最终花费比特币从帕洛阿尔托的鲍勃咖啡屋购买一杯咖啡。这个故事将从零售客户的角度介绍比特币软件、比特币的兑换，以及基本的交易过程。

北美高价值零售

卡罗尔（Carol）是旧金山一家画廊的老板，她以比特币计价出售昂贵的画作。这个故事将介绍高价值零售商面临的51%共识攻击的风险。

离岸合同服务

鲍勃（Bob），帕洛阿尔托一家咖啡店的老板，正在建设一个新的网站。他与一个居住在印度班加罗尔的网站开发工程师高佩什（Gopesh）签订了合同。高佩什同意鲍勃以比特币进行支付。这个故事将检验比特币在外包、合同服务及国际汇款中的使用。

慈善捐赠

尤金妮娅（Eugenia）是菲律宾一家儿童慈善机构的负责人。近来，她听闻了比特币的相关事宜，希望通过比特币接触一个全新的国内外捐赠群体，以支持她的慈善事业。她研究了如何利用比特币将善款分发到需要的地方去。这个故事将展示利用比特币进行跨币种和跨国界的筹款过程，以及利用开放账本实现慈善组织的透明化。

进出口

穆罕默德（Mohammed）是迪拜的一个电子产品进口商。他希望利用比特币从美国和中国进口电子产品到阿联酋，以加快进口的支付过程。这个故事将展示比特币是如何用于大型商业机构间基于物理货品的国际支付的。

比特币挖矿

景（Jing）是上海的一个计算机工程专业的学生。他建了一个矿机在比特币网络中挖矿，利用他的专业特长增加收入。这个故事将检验比特币的“产业”基础：专业的设备用于保护比特币网络，并发行新的货币。

以上每个故事均基于那些正在利用比特币创造新的市场、新的产业，提出创新方案以解决全球经济问题的真实人物和真实产业。

新手入门

要加入比特币网络并开始使用这种货币，用户首先要下载软件或者利用现成的web（网页）应用。由于比特币是一种标准，所以存在很多的比特币客户端。当然，也有一个标准程序，被称为中本聪客户端，它作为开源项目由一个开发团队进行维护，是从中本聪开发的原型中发展而来的。

主要有3种形式的比特币客户端：

完全客户端

完全客户端，或者被称为“完全节点”，是一个保存全部比特币交易历史（包含每个用户的每笔交易）的客户端，它管理用户的钱包，也可以直接在比特币网络中开始一笔交易。这种客户端类似于一个独立的e-mail（电子邮箱）服务器，不需要依赖任何其他服务器或第三方服务器就可以处理协议的方方面面。

轻量级客户端

轻量级客户端保存用户的钱包，它依赖于第三方服务器访问比特币交易和网络。轻量级客户端不保存完整的交易，所以，它必须信任第三方服务器以进行交易验证。就像电子邮箱的客户端，连接到一个电子邮箱服务器上访问邮箱，通过第三方与整个网络进行交互。

web客户端

通过浏览器访问web客户端，用户的钱包保存在一个第三方的服务器上。就像webmail（网页邮件），完全依赖第三方服务器。

移动客户端

在手机上运行的移动客户端，比如那些基于安卓系统（Android System）的客户端，可以是完全客户端，也可以是轻量级客户端或者web客户端。一些移动客户端与web或桌面客户端同步，提供一个基于相同资金来源的跨设备的多平台钱包。

客户端的选择基于客户对资金的控制意图。完全客户端提供了最高级别的独立控制，但是它也给用户带来了备份和安全方面的负担。web客户端最容易安装和使用，但是由于安全和控制是与web服务提供方共享的，这就带来了交易风险。如果web钱包服务被盗用（实际上已经发生过多次），用户就会失去他们的所有资金。相反地，如果一个用户拥有完全客户端，却没有进行足够的备份，他们也可能会因为电脑故障而丢失自己的资金。

基于本书的目的，我们将从标准程序（中本聪客户端）到web钱包，演示各种可以下载的客户端。某些例子需要参考客户端，它不仅是一个完整的客户端，也向我们暴露了钱包、网络和交易服务的API（应用程序接口）。如果你准备研究比特币系统的编程接口，参考客户端是必须的。

快速入门

爱丽丝，我们在“比特币的使用、用户，以及他们的故事”中介绍过的那位，她并不是一个技术宅，只是从朋友那里听说了比特币。她通过访问官方网站bitcoin.org开始了她的比特币之旅，在那儿，她看到了一个包含很多客户端列表的多个选项。根据该网站的建议，她选择了一个叫Mutibit的客户端。

爱丽丝根据bitcoin.org的下载链接，在自己的电脑上下载并安装了Multibit。Multibit提供了Windows、Mac OS和Linux桌面等多个版本。

比特币钱包必须用密码或口令进行保护。有很多不法分子会尝试破解脆弱的密码，所以必须小心地选择一个无法轻易破解的密码。尽量使用混合了大小写字母、数字和其他符号的密码。避免使用生日、运动队名称等与个人相关的信息。也要避免那些很容易在字典中找到的任何语言的词语。如果可能，采用密码生成器生成一个完全随机的、至少12个字符长度的密码。记住：比特币是钱，而且可以在瞬间转移到世界上的任何地方。如果不好好进行保护，它是很容易被人窃取的。

当爱丽丝下载并安装了Multibit软件后，她点击运行，出现了一个欢迎界面，如图1.1所示。

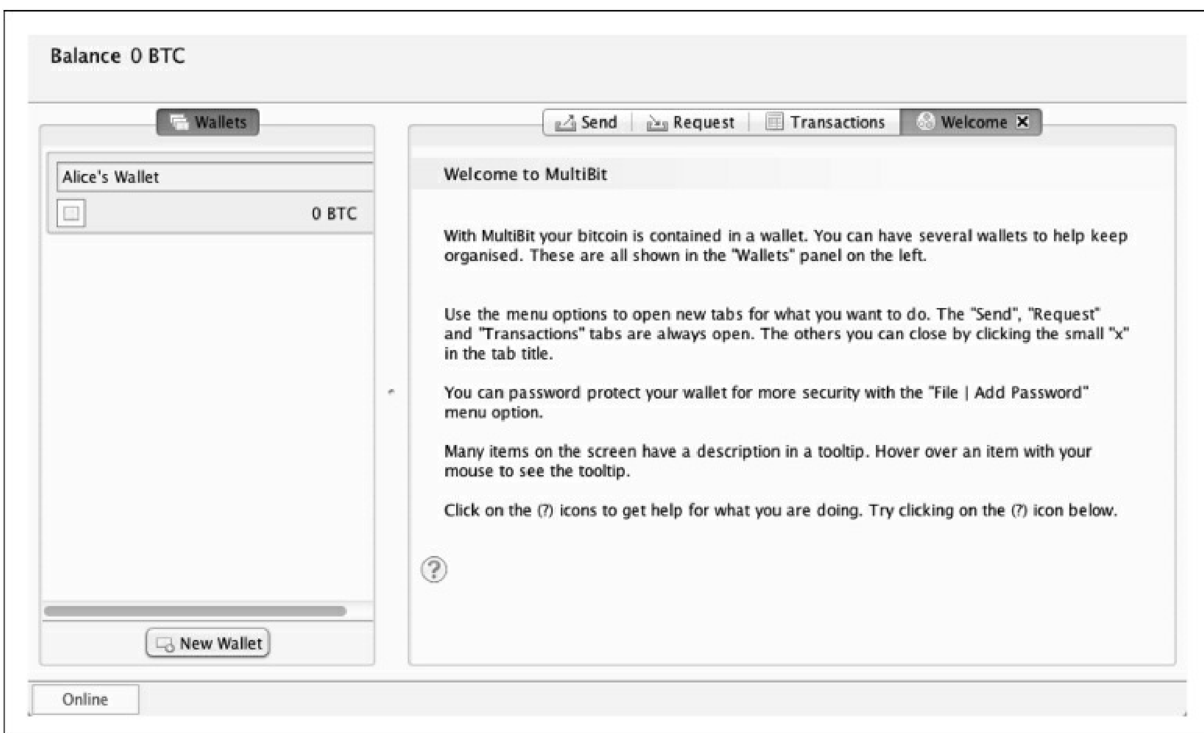


图1.1　Multibit比特币客户端欢迎界面

通过点击Request选项卡，可以看到Multibit已自动为爱丽丝创建了一个钱包和比特币地址。如图1.2所示。

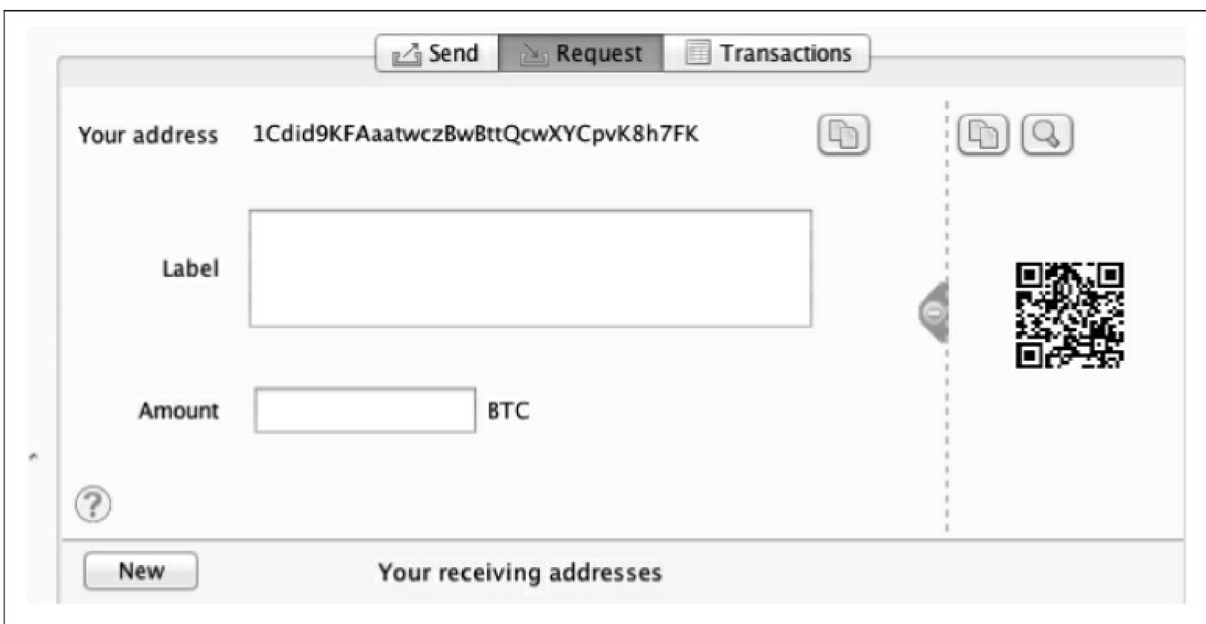


图1.2　在Multibit客户端的Request（请求）选项卡下的爱丽丝的新比特币地址

屏幕中最重要的部分是爱丽丝的比特币地址，就像一个电子邮箱地址，爱丽丝可以将这个地址分享给别人，任何人也可以利用这个地址直接给爱丽丝转钱，转完后钱将直接进入爱丽丝的新钱包。屏幕上，地址看起来是一长串由字母和数字组成的字符串：1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK。比特币地址旁边是一个二维码，它是另一种包含了相同信息的表现形式，可以被智能手机扫描识别。二维码就是窗口右方黑白图案构成的方框。爱丽丝可以通过点击比特币地址或者二维码边上的复制按钮将比特币地址或者二维码复制至剪贴板。如果单击二维码本身，它将放大显示以方便智能手机的摄像头扫描。

爱丽丝也可以将二维码打印下来，方便地交给其他人，免得他们输入那串冗长的字母和数字。

比特币地址开始于1或3。就像电子邮箱地址一样，可以在比特币用户间分享，别人可以通过这个地址给你的钱包直接发送比特币。与电子邮箱地址不同的地方是，只要你愿意，你可以随时创建新的比特币地

址，所有这些地址都可以让收到的资金直接进入你的钱包。钱包就是简单的一堆地址和它们内含的解锁资金的密钥的集合。你可以通过每笔交易采用一个不同的地址来提高交易的隐私性。事实上，一个用户能创建的地址数量是没有限制的。

爱丽丝现在已经准备好使用她的新比特币钱包了。

获取第一笔比特币

现在还没有办法从银行或者外币兑换处买到比特币。截至2014年，在大多数国家购买到比特币都还是很困难的。你可以去一些专业的货币交易所，在那儿，你可以以一定的汇率利用当地货币买卖比特币。这些交易所的运营模式通常都是基于web的，主要包括如下。

Bitstamp（http://bitstamp.net）

一个欧洲货币市场，它支持欧元（EUR）和美元（USD）等几种货币，通过电子转账的方式完成。

Coinbase（http://www.coinbase.com）

一个位于美国的比特币钱包和交易平台，在这个平台上，商家和客户可以通过比特币进行交易。Coinbase允许用户通过自动清算系统（ACH系统）将交易所账户与美元支票账户相连接，使得比特币的买卖变得非常简单。

这些加密货币交易所，是法定货币和加密货币的交汇点。它们同样需要符合国内国际的监管要求，并且经常被限定在一个国家或者经济区内，且指定使用该地区的法定货币。你指定的货币交易所只能使用你所在国家的货币，并且只能位于你的国家拥有司法管辖权的地方。与银行开户类似，你可能需要花上几天到几周的时间来开设使用这项服务的账

户。他们要求你提供各种形式的身份证明以满足KYC（了解你的客户）和AML（反洗钱）等银行监管规定。一旦你拥有了一个比特币交易所账户，你就可以像使用代理账户买卖外币一样开始买卖比特币了。

Bitcoin Charts（**http://bitcoincharts.com/markets**）是一个提供价格行情及大量货币交易所市场数据的网站，在这里你可以找到更完整的列表。

以下是作为新用户获得比特币的其他4个渠道：

●找到一个拥有比特币的用户，直接从他那里买一些。这也是很多新用户最初获取比特币的方法。

●使用类似localbitcoins.com的分类服务网站，找到一个当地的卖家，并通过当面交易的方式，用现金从他手里购买比特币。

●以比特币计价的方式出售一项商品或服务。如果你是个程序员，可以出售你的编程技术服务。

●利用你所在城市的比特币自动取款机（ATM）。利用CoinDesk（http://www.coindesk.com/bitcoin-atm-map）的在线地图在你附近找到一个比特币ATM。

爱丽丝是通过朋友介绍认识比特币的，因此当她在等待加州货币市场对她的账户进行验证和激活的过程中，也有很方便的途径可以获取她的第一笔比特币。

发送和接收比特币

爱丽丝已经创建好了比特币钱包，现在她可以接收资金了。她的钱

包软件随机生成了一个私钥（细节参看第4章“私钥”）及相应的比特币地址。这时候，她的比特币地址并没有让比特币网络知道，也没有注册到比特币系统的任何部分。她的比特币地址仅仅是一个数字，它与控制资金的密钥相关联。没有账户，也没有地址与账户关联。直到作为交易的价值接收方将该比特币地址发布到比特币账本（区块链）前，该地址只是无数可能的比特币“有效”地址的一部分。一旦它和交易相关联，它就变成了网络中已知地址的一部分，而爱丽丝也就可以在公共账本上查看它的余额了。

爱丽丝在一个当地餐馆碰到她的朋友乔（Joe），就是他介绍爱丽丝进入比特币世界的，爱丽丝用美元跟乔交换了一些比特币。她带着打印好的比特币地址和二维码作为她钱包的标识。从安全的角度看，比特币地址无关敏感信息。它可以发到任何地方而不用担心对她的账户安全造成威胁。

爱丽丝希望换10美元的比特币，她不想在这项新技术上冒太多金钱上的风险。她给了乔10美元的现金以及打印出来的地址。然后乔就可以给她相应金额的比特币了。

接着，乔需要查出当前的汇率，从而把正确金额的比特币发给爱丽丝。有很多软件和网站提供当前的市场汇率，以下是几个比较流行的：

Bitcoin Charts（http://bitcoincharts.com）

一个市场数据服务网站，提供全世界范围内多个交易所的、以本地货币标价的比特币市场汇率。

Bitcoin Average（http://bitcoinaverage.com/）

一个提供每种货币的“容量—权重—均值”概览的网站。

ZeroBlock（http://www.zeroblock.com/）

一个免费的安卓和iOS应用，可以显示不同交易所的比特币价格。参见图1.3。

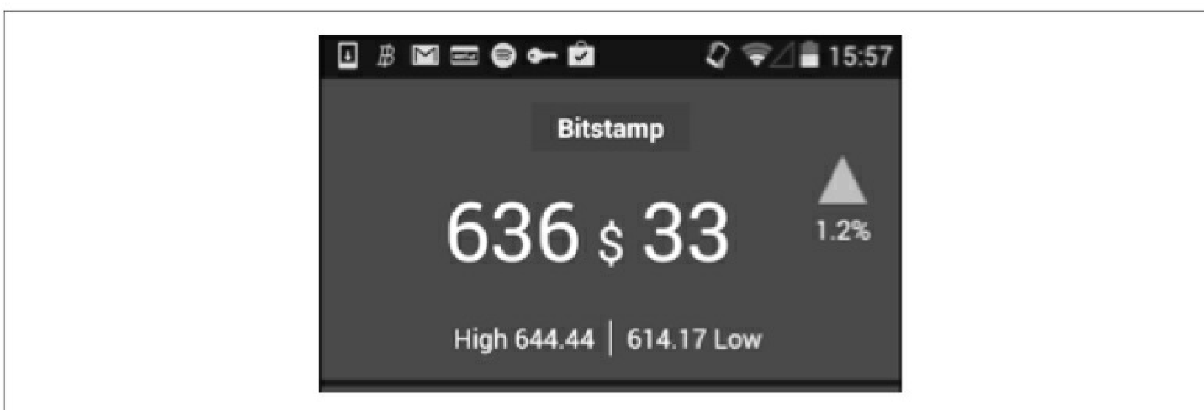


图1.3　ZeroBlock，一个适用于安卓、iOS系统的比特币市场汇率应用

Bitcoin Wisdom（http://www.bitcoinwisdom.com/）

另一个市场数据服务网站。

使用以上某个网站或者应用，乔确定了每个比特币的价格约为100美元。在这个汇率下，作为爱丽丝给他的10美元的对价，他需要付给爱丽丝0.10个比特币，或者100毫币。

当乔确定了合理的交换价格后，他打开他的移动钱包应用，选择发送比特币。比方说，在安卓手机上选择“Blockchain（区块链）”移动钱包，他将看到屏幕上有两个必需的输入项，如图1.4所示。

●交易目标地址。

●待发送比特币金额。

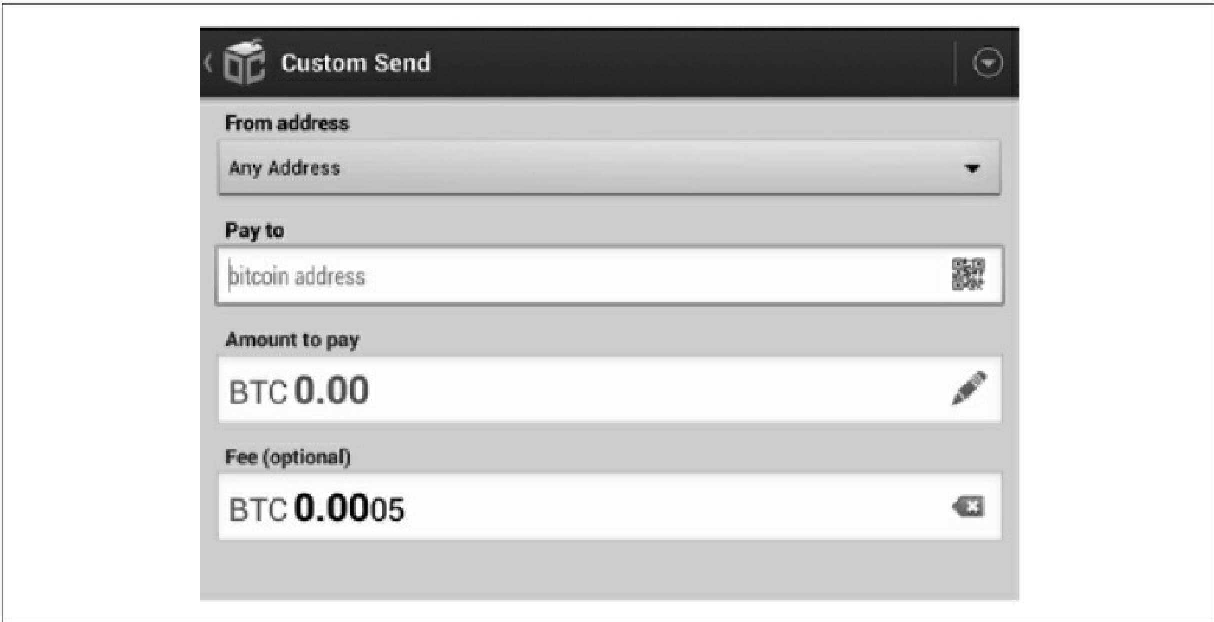


图1.4　区块链移动钱包的比特币发送界面

在地址输入域中，有个类似二维码的小图标。它允许乔利用他的手机摄像头扫描二维码，这样就避免了手工输入爱丽丝的那串又长又难输的比特币地址（1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK）。乔点击二维码图标，激活手机摄像头，从爱丽丝给他的打印好的纸张上扫描到了二维码，并自动填充到比特币地址字段上。乔可以从中挑几个字符与爱丽丝打印出来的地址进行比对，以确认是否正确。

然后乔输入比特币交易金额：0.10比特币。他认真地进行了检查以确定输入了正确的金额，因为他准备做的是真正的金钱的传递，任何错误都会带来“实打实”的损失。最后他按下了发送按钮，将交易指令发送出去。这样，乔的移动比特币钱包就创建了一个交易，这个交易将发送0.10比特币给爱丽丝提供的地址，资金的来源是乔的钱包，采用乔的私钥进行了签名。这告诉比特币网络，乔已经授权从他的地址转账0.10比特币到爱丽丝的新地址。随着交易通过点对点协议进行传输，它很快就广播到了比特币网络中。不到1秒，大部分有良好连接的节点已经接收到了交易，并且第一次看到了爱丽丝的新地址。

如果爱丽丝随身带着智能手机或者笔记本电脑，她也能看到这笔交易。比特币账本（一种持续增长的，记录所有已发生的比特币交易的文件）是公开的，意味着她只需要关注她自己的地址，查看是否有资金发送到该地址。她可以在blockchain.info网站上输入她的地址，很容易就能做到这点。网站会向她展示一个页面（**http://bit.ly/1u0FFKL**），这个页面包含了从这个地址发出或接收到的所有交易的信息。如果爱丽丝正在查看这个页面的话，在乔点击发送后，页面很快就会显示出最新的一笔乔转给她0.10比特币的交易信息。

确认

最初，爱丽丝的钱包会显示乔发过来的交易处于“未确认”状态。意思是交易已经广播到网络，但是还没有被包含在比特币交易账本（被称为“区块链”）中。为了被交易账本包含进去，交易需要被矿工挑出，并打包进一个交易区块中。在差不多10分钟内，当一个新的交易区块被创建出来时，已包含入区块的交易会被网络接受为“已确认”状态，这笔钱已经可以使用了。交易可以立即就被看到，但是只有被包含进新挖出来的区块中，才能得到所有人的“信任”。

爱丽丝现在是光荣的比特币拥有者了，她可以使用0.10比特币。在下一章，我们将看到她使用比特币的第一次购买行为，并将更细致地了解交易底层及交易传播技术。

第**2**章 比特币是如何工作的

交易、区块、挖矿和区块链

比特币系统不像传统银行系统或支付系统，它基于一种去中心化的信用。与传统的中央权威信用机构相反，比特币的信用是系统的一种自然属性，它源于比特币系统不同参与者间的交互。在本章中，我们将从较高的层面审视比特币系统，通过跟踪一笔交易，看它是如何在分布式共识机制下变为“被信用”和被接受，并最终被记录到区块链（记录所有交易的分布式账本）上的。

书中的每个例子都是在比特币网络上真实执行过的交易，模拟了资金在用户（乔、爱丽丝、鲍勃）间转移的交互过程，即比特币从一个钱包进入另一个钱包的过程。为了在比特币网络和区块链中跟踪交易，我们可以利用区块链浏览器网站来可视化地展现每个步骤。区块链浏览器是一个web应用，起到了比特币搜索引擎的作用，通过它可以搜索地址、交易、区块，并查看它们的关系和流程。

常用的区块链浏览器包括：

●Blockchain info（http://blockchain.info）。

●Bitcoin Block Explorer（http://blockexplorer.com）。

●insight（http://insight.bitpay.com）。

●blockr Block Reader（http://blockr.io）。

以上每个网站的区块链浏览器均带有搜索功能，可以依据地址、交易哈希、区块号码进行搜索，每个网站的搜索结果在比特币网络和区块链中都是等效数据。在每个例子中，我们都会提供一个URL，让你直接

找到相应的入口，以便对细节进行进一步研究。

比特币概览

在图2.1中，我们可以看到：比特币系统由用户、交易和矿工组成，用户拥有钱包，钱包中管理着用户的密钥；交易在网络中传播；矿工通过竞争性计算来创建共识区块链，它是所有交易的权威账本。在本章中，我们将从较高的层面，跟踪一个交易在网络中的传播过程，并检视由此引起的比特币系统不同部分间的交互。接下来的章节中，我们将深入了解钱包、挖矿和商户系统背后的更多技术细节。

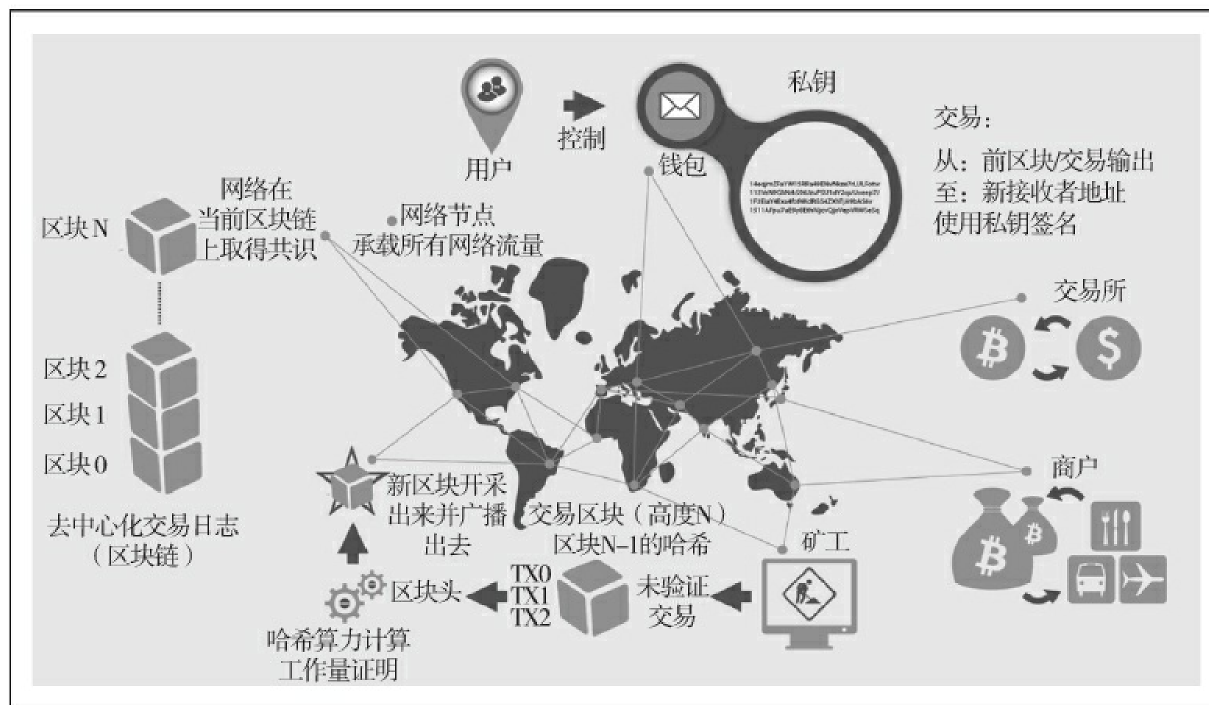


图2.1　比特币概览

购买一杯咖啡

爱丽丝，这位我们之前已经介绍过的主角之一，是一个新用户，她

刚刚获得了她的第一笔比特币。在第1章“获取第一笔比特币”中，爱丽丝遇到了她的朋友乔，并用现金跟他交换了一笔比特币。那笔由乔创建的交易往爱丽丝的钱包中存进了0.10比特币（BTC）。现在爱丽丝将进行她的第一笔基于比特币的零售交易——在鲍勃位于加州帕洛阿尔托的咖啡店购买一杯咖啡。鲍勃的咖啡店最近刚开始接受比特币支付，他在销售系统上添加了比特币的选项。咖啡店的价格是以当地货币（美元）标注的，但是在收银处，客户可以选择以美元或者比特币进行支付。销售系统将自动依据主流市场的汇率，将美元价格转换为比特币价格，并同时显示两种价格，销售终端还会在屏幕上显示一个带有该笔支付请求的二维码，如图2.2所示。

```
Total:
$1.50 USD
0.015 BTC
```



图2.2　支付请求的二维码（提示：扫扫看！）

支付二维码按照BIP0021的要求，将以下URL进行编码：

```
bitcoin:1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA?
amount=0.015&
label=Bob%27s%20Cafe&
message=Purchase%20at%20Bob%27s%20Cafe
```

```
Components of the URL

A bitcoin address: "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA"
The payment amount: "0.015"
A label for the recipient address: "Bob's Cafe"
A description for the payment: "Purchase at Bob's Cafe"
```

不像一个只包含目标地址的二维码，支付请求二维码是一个以二维码格式编码的URL，包含了目标地址、支付金额、通用的交易描述，如“鲍勃咖啡店”。比特币钱包软件可以将这些信息预填充，用于发送支付指令的信息，同时也能以人工可读的方式显示给用户。你可以使用一个钱包软件扫描二维码，看看爱丽丝看到了什么。

鲍勃说：“共1.5美元或者15毫比特币。”

爱丽丝用她的智能手机在屏幕上扫了一下二维码，手机显示需要支付0.0150比特币给鲍勃咖啡店，她点击“发送”，授权了这笔支付交易。几秒钟后（与刷信用卡授权的时间差不多），鲍勃在收银机上看到了这笔交易，交易完成了。

在接下来的章节中，我们将介绍这笔交易的细节，了解爱丽丝的钱包如何构建这笔交易，如何将它传播到网络中，交易如何获得确认，最后鲍勃怎么才能在后续交易中使用这笔钱。

比特币网络支持将比特币拆分成小额进行交易，从千分之一比特币（一毫比特币）到一亿分之一比特币（通常被称为“聪”，satoshi）都可以。在本书中，我们用“比特币”指代任何数量的比特币货币，从最小的单位（1聪）到全部可被挖出的数量（2100万）。

比特币交易

简单来说，一个比特币交易，就是告诉网络，某个拥有一定数量比特币的用户已经授权将这笔比特币转让给另一位用户。新的所有者可以通过另外一笔授权转让交易来使用这些比特币，以此类推，形成一个所有者转换的链条。

交易就像复式账本的一笔笔记录，每个交易均包含一到多条的“输入”——这是比特币账户的借方。每笔交易也包含了一到多条的“输出”——这是比特币账户的贷方。输入和输出（借和贷）加起来不要求相等。实际上，输出加起来的和应稍小于输入的和，这个差额就是隐含的“交易费用”，这笔小额费用归那些将交易归集到账本的矿工所有。以复式记账法表示的比特币交易如图2.3所示。

采用复式记账法表示的交易

输入	价值	输出	价值
输入 1	0.10 BTC	输出 1	0.10 BTC
输入 2	0.20 BTC	输出 2	0.20 BTC
输入 3	0.10 BTC	输出 3	0.20 BTC
输入 4	0.15 BTC		
输入合计：	0.55 BTC	输出合计：	0.50 BTC

	输入	*0.55 BTC*
–	*输出*	*0.50 BTC*
	差	*0.05 BTC（隐含交易费用）*

图2.3　采用复式记账法表示的交易

交易同样包含每笔待转让比特币（交易输入）的所有权证明，以所有者数字签名的方式来表示，数字签名可以被任何人独立验证。在比特币的术语中，“消费”就是签署一笔交易，将所有者从前序交易中获得的价值权益转让给以比特币地址为代表的新所有者。

交易将价值从交易输入转移到交易输出。交易输入是价值的来源，通常是上一笔交易的输出。交易输出将一笔与私钥关联的价值赋予一个新用户。目标密钥称为安全锁。在未来的交易中，需要通过签名来获取这笔资金。一笔交易的输出作为新交易的输入，这样，随着价值不断从一个地址转移到另一个地址，就形成了一条所有权的链条（见图2.4）。

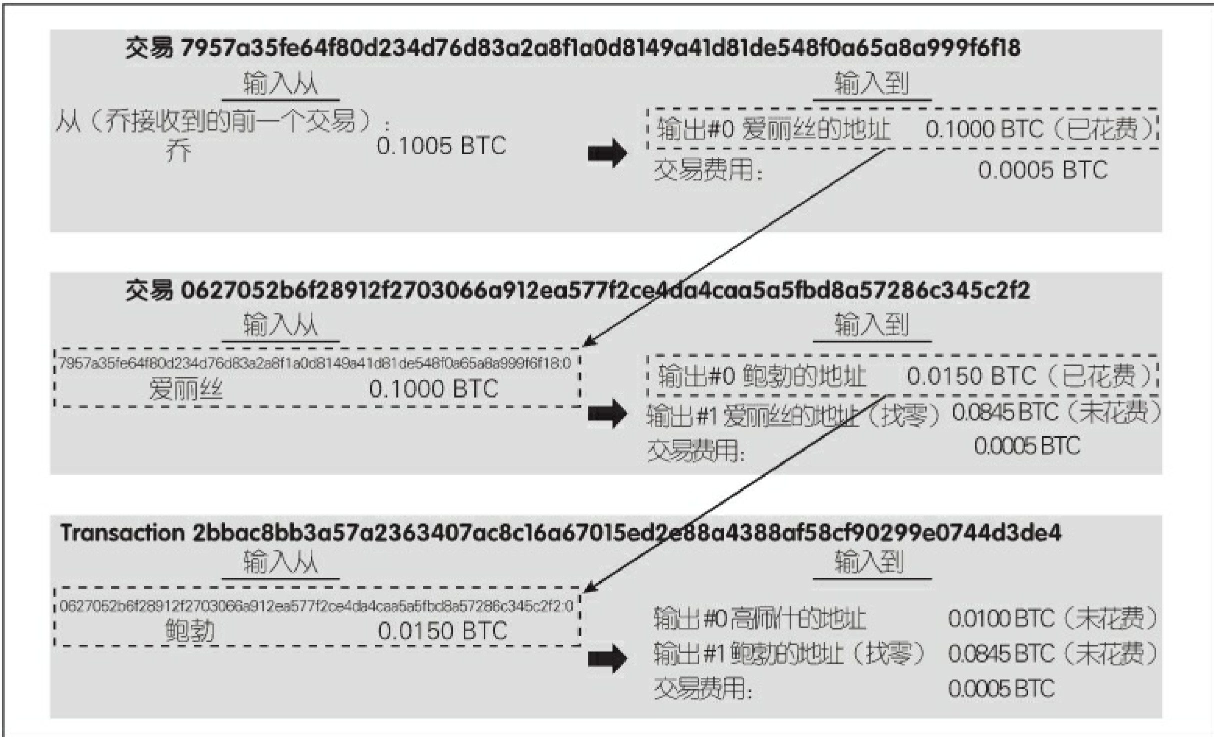


图2.4　一个交易的链条，在链上一个交易的输出成为下一笔交易的输入

爱丽丝支付给鲍勃咖啡店的交易使用了上一笔交易的输出，作为这笔交易的输入。在上一章中，爱丽丝用现金从她的朋友乔那里换到了一笔比特币。那个交易的一笔资金被爱丽丝的密钥锁定（受限）。她向鲍勃咖啡店支付咖啡费的新交易中，引用了上笔交易的输出作为本笔交易的输入，输出则包含两部分，一部分支付咖啡费用，另一部分用于找零。交易形成了一个链条，最新交易的输入对应上一笔交易的输出。爱丽丝的密钥提供的签名解锁了前笔交易的输出，向比特币网络证明了她对这些资金的所有权。她在交易中附上鲍勃的地址，形成一个“受限”，限制鲍勃必须使用签名才能花费这笔资金。这个过程展示了价值在爱丽丝和鲍勃之间转移的过程。这个从乔到爱丽丝，再到鲍勃的交易链见图2.4。

常见交易形式

最常见的交易就是从一个地址到另一个地址的支付，它通常还包含需要返还给初始所有者的零钱。这种类型的交易包含一个输入和两个输出，参见图2.5。

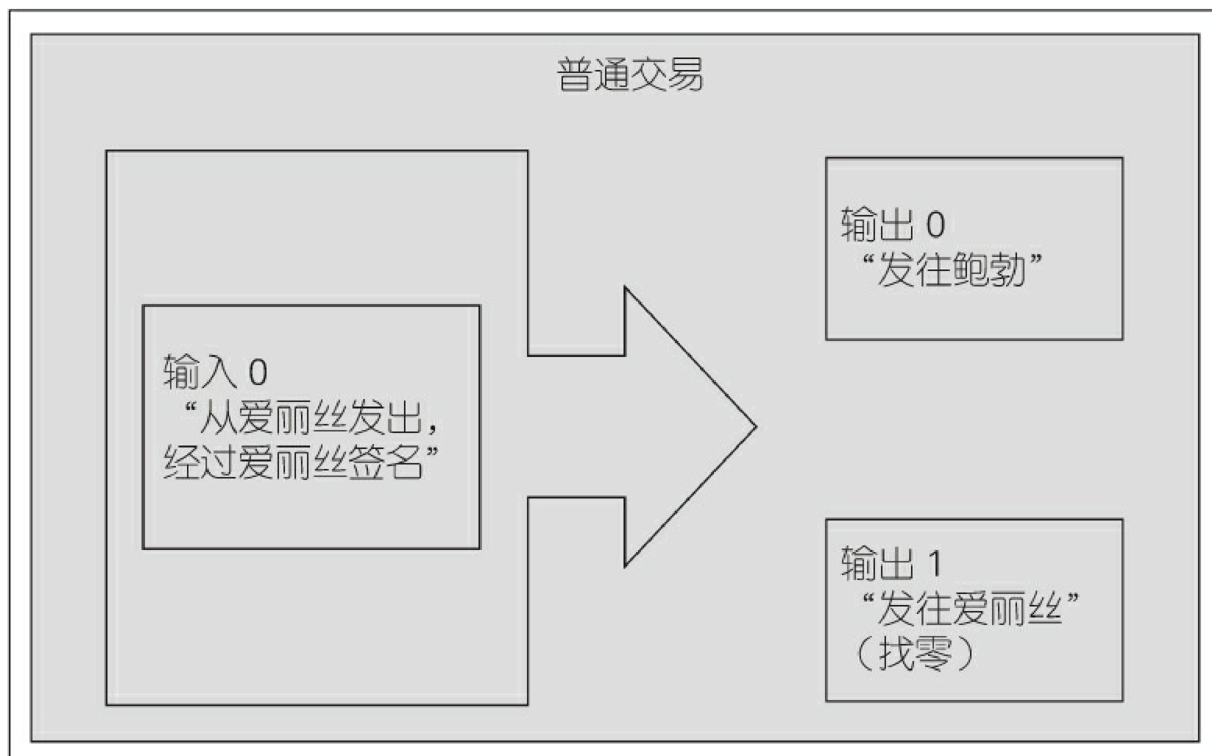


图2.5 最常见的普通交易

另一种常见交易形式是绑定几个输入，形成一个输出（见图2.6）。这与现实生活中将零钱换为大额钞票的场景类似。这种交易通常是钱包软件对交易中找回的一堆零钱进行清理的情况。

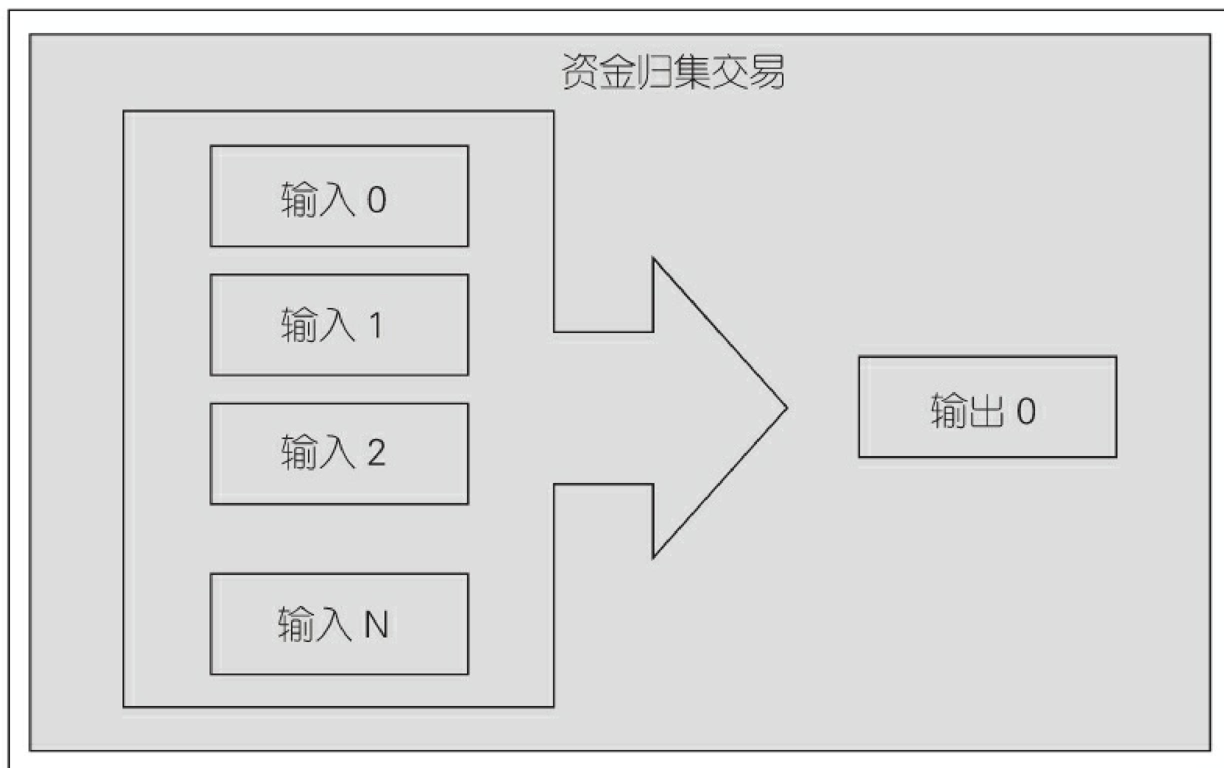


图2.6　资金归集交易

最后，还有一种在比特币账本中常见的交易形式，即将一个交易输入分配给多个输出，每个输出代表不同接收者（见图2.7）。这种类型的交易有时是商业实体用于分配资金，比如给员工发放工资。

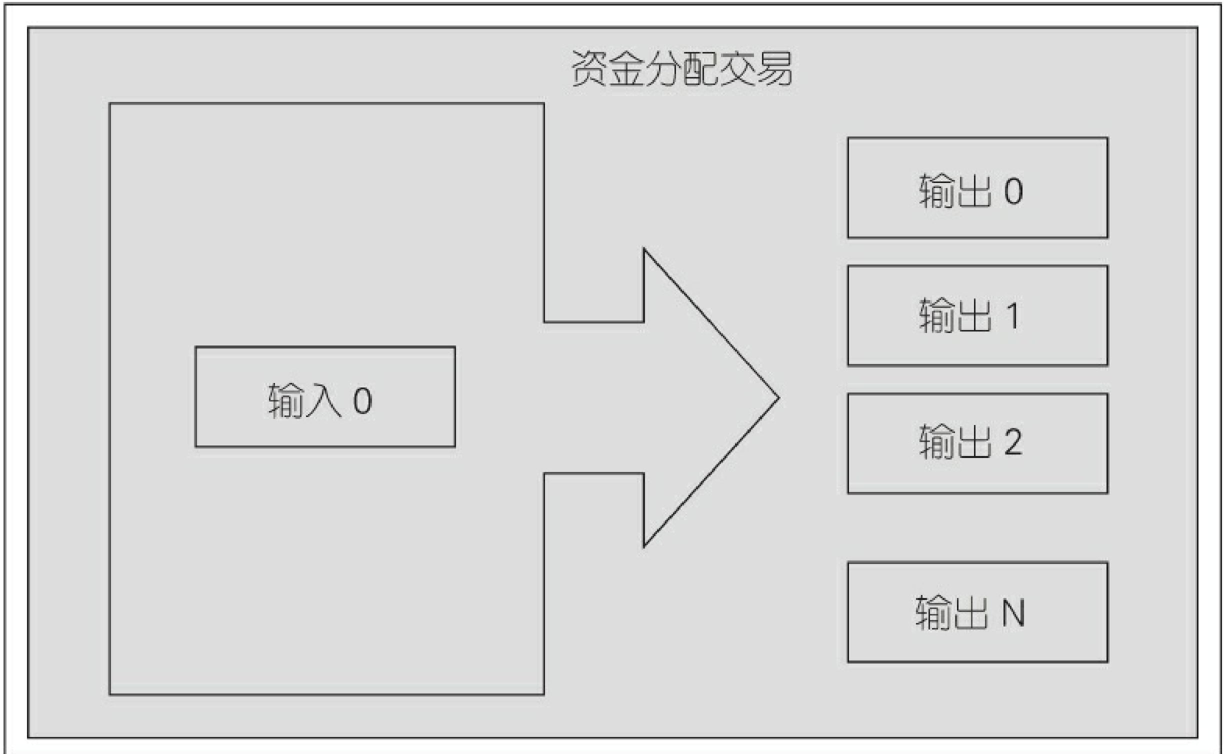


图2.7　资金分配交易

创建交易

钱包软件知道如何选择合适的输入和输出，以创建符合爱丽丝要求的交易。爱丽丝要做的只是指定一个接收人和交易金额，不用操心具体的细节，钱包软件会自动完成剩下的工作。很重要的一点是，钱包软件即使在离线的情况下也可以创建交易。就像在家里写一张支票，然后将其装进信封寄给银行，创建交易和签名不需要在连接到比特币网络的情况下进行，只有在执行交易时才需要将其发送到网络。

获得正确的输入

首先，爱丽丝的钱包软件必须找到足以支付鲍勃资金的交易输入。大部分钱包软件会保留一个“未花费的交易输出”的小型数据库，由钱包所有者的私钥锁定（受限）。因此，爱丽丝的钱包含有那笔她用现金跟乔交换比特币的交易输出的副本（参见第1章“获取第一笔比特币”）。一个运行在完全客户端下的钱包软件，实际上包含了网络上所有交易的“未花费的输出”。这使得钱包软件不仅能快速构建交易输入，也能验证一个新来的交易，判断其输入是否有效。由于完全客户端需要耗费非常多的存储空间，实际上大部分用户的钱包软件只运行一个轻量级客户端，只能用于跟踪用户自己的未花费输出。

如果一个钱包软件没有维护完整的“未花费的输出”，它可以使用不同供应商提供的API接口向比特币网络询问这些信息，也可以使用JSON RPC API接口向一个完全节点询问相关信息。例2-1展示了一个RESTful API请求，它以HTTP GET命令的方式构造一个请求发往特定的URL地址。这个URL将根据请求中提供的地址返回所有“未花费的输出”信息，供任何需要这些信息构建交易输入的应用使用。在这里，我们使用简单

的命令行模式的HTTP（超文本传输协议）客户端cURL来发送请求并获取应答。

例**2-1**　获取爱丽丝比特币地址下所有未花费的输出

```
$ curl https://blockchain.info/unspent?active=1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK
```

例**2-2**　请求应答

```
{

        "unspent_outputs":[

                {
                        "tx_hash":"186f9f998a5...2836dd734d2804fe65fa35779",
                        "tx_index":104810202,
                        "tx_output_n": 0,

"script":"76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",
                        "value": 10000000,
                        "value_hex": "00989680",
                        "confirmations":0
                }

        ]
}
```

例2-2的应答显示，在爱丽丝的地址1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK下有一笔“未花费的输出”（尚未兑现）。应答还包含了产生这笔输出的交易引用（乔发起的支付交易），以单位“聪”表示的交易金额是1000万聪，也就是0.10比特币。利用这些信息，爱丽丝的钱包软件就可以构建出一个新交易，向新地址发送资金了。

在http://bit.ly/1tAeeGr查看从乔到爱丽丝的交易。

就像你看到的，爱丽丝的钱包软件中唯一的未花费输出已足够支付

一杯咖啡的费用。如果这个条件不能满足，钱包软件就不得不去翻找所有的小额未花费输出，凑够这笔交易输入，就像从手提袋中翻找足够的硬币来凑够一杯咖啡的钱。在以上两种情况下，钱包软件都可能需要在交易输出中找回一些零钱，我们将在下节讨论。

创建输出

交易输出以脚本的形式创建一个针对特定价值的受限，只有通过解决脚本问题来解除受限，才能兑现这笔输出。简单地说，爱丽丝的交易输出包含一个脚本，脚本大意是这样的：“这个输出将支付给那个能提供与鲍勃的公开地址相匹配的签名的人。”由于只有鲍勃拥有与其地址匹配的密钥，所以只有鲍勃的钱包软件可以提供这样一个签名来兑现这笔输出。爱丽丝以要求提供签名的方式，锁定了一笔输出价值。

这笔交易还包含另一部分，因为爱丽丝的资金是以0.10比特币表示的一个输出，超过了一杯咖啡0.015比特币的金额。爱丽丝需要拿回0.085比特币的找零。爱丽丝的找零操作是“爱丽丝的钱包软件”在生成给鲍勃的支付交易时一并产生的。本质上看，爱丽丝的钱包软件将她的资金分成两笔支付，一笔给鲍勃，剩下的交还她自己。她可以在下一笔交易中使用这个找零输出。

最后，为了使这笔交易尽快被网络执行，爱丽丝的钱包应用将添加一小笔的交易费用。这个金额不是显式的，它隐含于交易输入输出的差值中。假如交易中找零金额不是填0.085，而是用0.0845作为交易第二个输出（找零输出），这样就有0.0005比特币（半毫比特）的剩余。输入的0.10比特币没有被两个输出完全花费完，因为它们加起来不到0.10。输入输出的差额就构成了交易费用，矿工将这笔交易费用加入区块并放入区块链账本的过程中，会收集这些交易费用作为他们挖矿的报酬。

交易结果可以通过一种叫作区块链浏览器的web应用来查看，结果如图2.8所示。

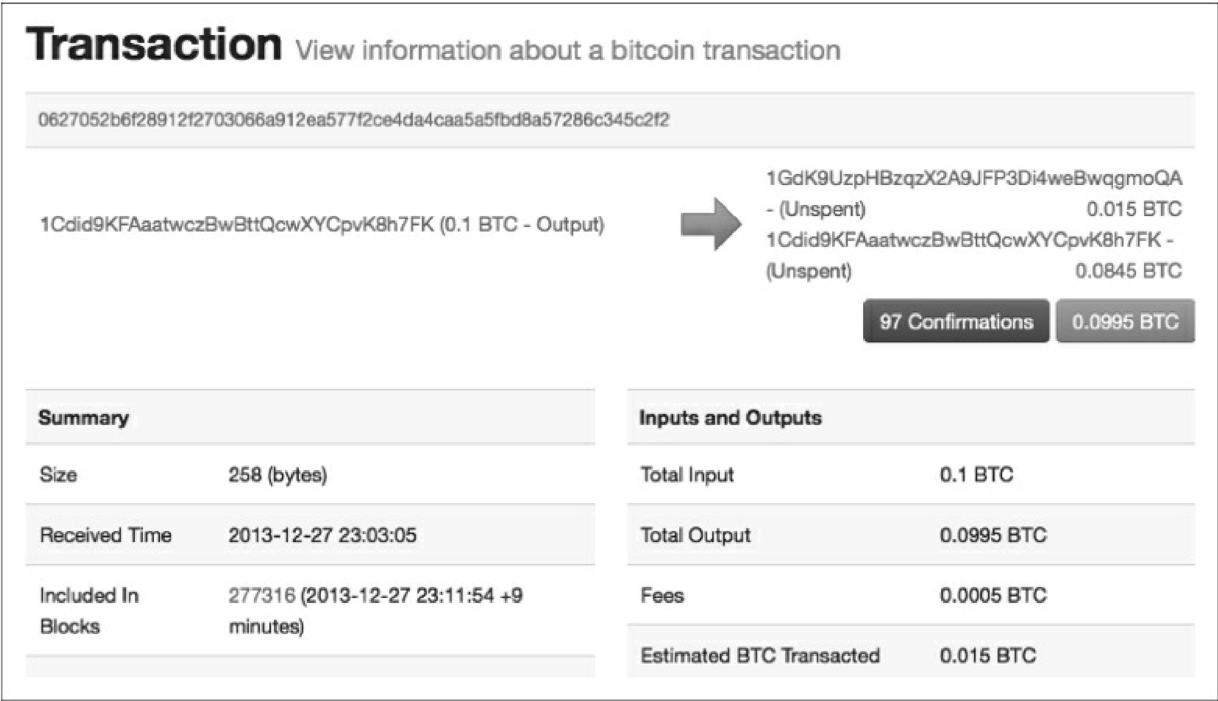


图2.8　爱丽丝给鲍勃咖啡店付款的交易

在http://bit.ly/1u0FIGs查看爱丽丝给鲍勃咖啡店付款的交易。

将交易添加到账本上

爱丽丝的钱包软件创建的交易共258字节长，它包含着用以证明爱丽丝的资金所有权的信息以及资金接收者的信息。现在，交易必须发送到比特币网络，并使之成为分布式账本（区块链）的一部分。在接下来的一节中，我们将看到一个交易如何变成新区块的一部分，以及区块是如何被“挖出来”的。最后，我们还将看到这个新区块被加入区块链后，是如何随着区块数量的增长而变得越来越可信的。

发送交易

因为交易已包含所有用于后续处理的信息，所以不必关心它是如何或者从哪里传入比特币网络的。比特币网络是一个点对点的网络，每个比特币客户端通过与不同的客户端连接，成为网络的参与者。比特币网络的目标就是把交易和区块传播给所有参与者。

如何传播

爱丽丝的钱包软件可以将新交易发送到所有通过互联网与它相连的客户端，不管它是通过有线网络、无线网络（Wi-Fi），还是通过移动网络相连的。她的比特币钱包不一定非要与鲍勃的钱包直接相连，她也不必非要使用咖啡店提供的互联网接入，虽然这两种方式也没什么不可以。任何比特币网络节点（其他客户端）接收到之前未见过的有效交易时，将立即将其转发给与它相连的其他客户端。这样，交易就很快在这个点对点网络中传播开来，在短短几秒内即可到达大部分节点。

鲍勃的视角

如果鲍勃的钱包软件与爱丽丝的直接相连，鲍勃的钱包软件将是第一个接收到交易的节点。不过，即使爱丽丝的钱包通过别的节点发送，交易也会在短短几秒内到达鲍勃的钱包。鲍勃的钱包会立即将爱丽丝的交易识别为一笔消费支付交易，因为交易含有需要鲍勃的密钥进行解锁的输出。鲍勃的钱包软件也能独立确认这笔交易是有效构建的，使用了之前未花费的输出，并且包含了足够的交易费用使其能够涵盖下个区块。基于此，鲍勃可以认定这个交易会很快被确认并添加到区块中，且交易的风险很小。

关于比特币的交易，有个常见的认识误区：认为交易必须等待10分钟被新区块“确认”，或者等待1小时得到全部6个确认后才是有效的。虽然确认可以确保交易被全网接受，但是这种延迟对于小额支付——比如一杯咖啡，其实是没有必要的。商户对这种小额交易可以直接

接受，其风险不会比一笔没有使用身份证或签名的信用卡交易风险更大，而现在商户一般都接受这种信用卡支付方式。

比特币挖矿

交易现在被传播到了网络中，在没有被确认并通过挖矿包含到区块前，它还没有成为共享账本（区块链）的一部分。关于挖矿，在第8章中将有详细介绍。

比特币系统的信用是建立在计算的基础上的。交易打包进区块需要巨大的计算量来证明，但是验证这个证明只需很少的计算量。挖矿过程在比特币系统中有以下两个目的：

●挖矿过程在每个新区块中创建新的比特币，就像中央银行发行货币。每个区块创建的新比特币数量是固定的，随着时间推移，这个数量会逐渐减少。

●挖矿过程创造信用，需要确保只有足够算力投入到包含这些交易的区块后，交易才能得到确认。更多的区块意味着更多的计算量投入，也意味着更多的信用。

挖矿的过程就像一个大型的竞争性数字拼图游戏，当有人找到一个解决方案时，游戏就重新开始，而游戏的难度也会自动进行调整，使每找到一个解决方案的时间大致维持在10分钟。想象一个巨大的数字拼图，高度有几千行，宽度有几千列。如果我给你看已经填充好的拼图，你可以很快地验证有没有错误。但是，如果只填了一部分，剩下的都是空白，那就需要花费大量的时间才能解决。数字拼图游戏的难度可以通过调整尺寸（增减行列数）来调节，但是不管尺寸大小，其确认过程都很简单。比特币中用的“拼图”是建立在加密哈希算法之上的，它展现了与拼图类似的特性：它也是不对称的，很难解决却很容易验证，而且它的难度也可以调整。

在第1章的“比特币的使用、用户，以及他们的故事”中，我们介绍过景，一个上海的计算机工程专业学生。景是以矿工身份参与到比特币网络中的。每10分钟左右，景与成千上万的矿工一起进行一场查找区块解决方案的全球竞赛。为了找到一个解决方案（被称为“工作量证明”），全网每秒要进行几万亿次的哈希计算。比特币中的工作量证明算法是采用SHA256加密哈希函数不断地对区块头和一个随机数进行哈希计算，直到找到一个与预设的模式匹配的方案。第一个找到这个解决方案的矿工将赢得这一回合的竞争，随即将这个区块发布到区块链当中。

景从2010年开始挖矿，他使用一台非常快的台式计算机来查找新区块的工作量证明。随着越来越多的矿工加入比特币网络中，挖矿的难度急剧增大。很快地，景和其他矿工将他们的电脑硬件升级到了更专业的水准，比如那些在游戏电脑或终端中使用的高端专用图形处理单元（GPU）。截至写这本书的时候，挖矿的难度已经非常非常高，为了保证有利可图，只能使用特定用途集成电路芯片（Application Specific Integrated Circuits，简称ASIC）来进行挖矿，这些ASIC芯片将几百个挖矿算法集成到硬件中，并使它们在一个芯片内并行计算。景加入了一个“矿池”，就像彩票池，允许多个参与者参与其中，共同工作并共享收益。景现在使用由两个通串线（USB）连接的ASIC机器，每天24小时挖矿，他通过出售挖矿获得的比特币来支付购买硬件的费用，并获取一定收益。他的电脑运行着一个bitcoind（标准比特币客户端）作为他的专用挖矿软件的后端。

交易区块挖矿

一个在网络上传播的交易，直到成为全局分布式账本（区块链）的一部分才算真正得到确认。平均每隔10分钟，矿工就会创建一个包含上个区块以来所产生的所有交易的新区块。新交易不停地从用户的钱包或者其他应用中流入网络。当这些交易被其他节点捕获时，就会被加入一个各自维护的临时未验证交易池中。矿工创建新区块时，他们将未验证交易池中的交易取出，并入新建区块，然后尝试解决一个极为困难的问题（即工作量证明）来证明这个区块的有效性。挖矿的过程我们将在第8章中详细说明。

根据费用优先原则及其他一些规则，交易被顺序加入新区块中。当矿工从网络中接收到上一个区块时，他会立即发现自己已经在上一轮竞争中失败了，所以立即开始新区块的挖矿过程。矿工首先创建一个新区块，填上交易以及上个区块的指纹，然后开始计算这个新区块的工作量证明。矿工还会在区块中包含一个特殊的交易，这个交易向他自己的比特币地址发送一笔新创建的比特币作为奖励（当前每区块25比特币[注]）。如果找到一个工作量证明使区块有效，他就赢得了这个奖励，因为他挖出的区块被成功加入全局区块链中后，他加入的奖励交易也变得可用了。由于加入了矿池，景把挖矿软件的新区块奖励地址设置为矿池的地址。在矿池中，一旦在上一轮挖矿竞争中胜出，由此获得的奖励将按照矿工所贡献工作量的大小进行分配。

爱丽丝的交易被网络节点提取并放进未验证交易池。因为交易包含了足够的费用，它会被放进景所在矿池的新建区块当中。交易从爱丽丝的钱包提交后大概5分钟，景的ASIC矿机找到了这个区块的工作量证明，并将其发布为第277316号区块，这个区块还包含419个其他交易。随着新区块在网络中的发布，其他矿工将立即对其进行验证并开始新一

轮的挖矿竞赛，以生成下一个区块。

你可以通过**http://blockchain.info/block-height/277316**查看包含爱丽丝交易的区块。

几分钟后，第277317号区块又被别的矿工挖出。由于这个新区块基于上一个包含了爱丽丝交易的区块（277316号区块），它在原有区块的基础上进行了更多的计算，因此进一步强化了对那些交易的信用。包含爱丽丝交易的区块被认为是对该笔交易的一个确认。基于这个区块，每产生一个新区块，就是对交易的一次额外确认。由于新区块一个个叠加在原有区块之上，这使得推翻原有交易的难度呈指数级增长，这样就保证了交易可信程度越来越高。

在图2.9中，我们可以看到包含爱丽丝交易的第277316号区块。在它之下有277316个区块（包括0号区块），这些区块互相连接，直到0号区块——被称为创世区块，形成一个区块的链表（区块链）。随着时间的推移，区块的“高度”不断增长，每个区块及整个链表的计算难度都不断增加。在这个越来越长的链表上，不断叠加新的计算，包含着爱丽丝交易的区块之后所挖出的区块成了该笔交易的额外保证。按照惯例，任何经过6次确认后的区块即被认为是不可撤销的，因为要撤销并重新计算6个区块需要极大的计算量。我们将在第8章详细介绍挖矿的过程以及它创建信用的机制。

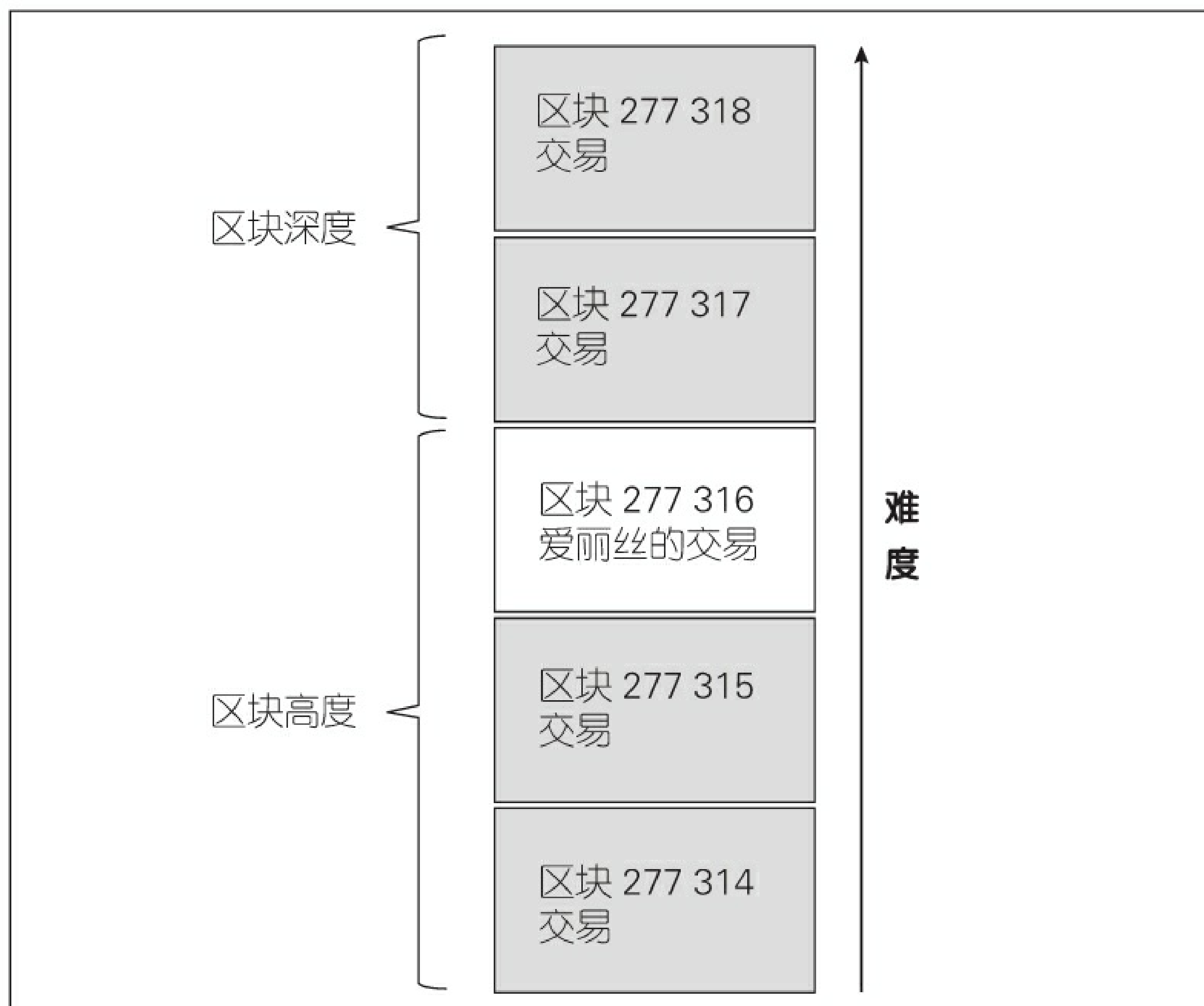


图2.9　爱丽丝的交易包含在第277316号区块中

---

1. 此为作者写书时的奖励金额，翻译此书时是12.5比特币。——译者注

花费交易

既然爱丽丝的交易已经作为区块的一部分被包含进区块链中，它就成了比特币分布式账本的一部分，对所有比特币应用都是可见的。每个比特币客户端都可以独立验证这笔交易，确认它是有效且可花费的。完全客户端可以从比特币在区块中创建的时刻起，跟踪这笔资金的流转过程，直到它们到达鲍勃的地址。轻量级客户端只能确认交易是否在区块链中，有几个区块在它之后被挖出，从而获知网络已确认它的有效性并接受它。轻量级客户端的这种操作被称为简化支付验证，参看第6章“简化支付验证节点”。

鲍勃现在可以引用这笔及其他笔交易的输出作为交易输入，生成他自己的新交易，并将资金转给新的所有者。举例来说，鲍勃可以用爱丽丝买咖啡的钱来支付承包商或供应商的费用。大多数情况下，鲍勃的比特币软件需要归集多个小的支付交易才能完成一笔较大的支付，或是将一天的比特币收入集中到一个交易中。这将把不同的支付交易的输出归集到一个作为咖啡店的“经常”账户使用的地址下。参看图2.6了解整合交易。

当鲍勃将从爱丽丝及其他客户收到的款项花出去的时候，他扩展了交易链条。结果是，新的交易加入全局区块链账本中，所有人都可见并获得信用。我们假定鲍勃向他的网站设计师——班加罗尔的高佩什支付新网页的设计费用。现在交易链条看起来就像图2.10这样。

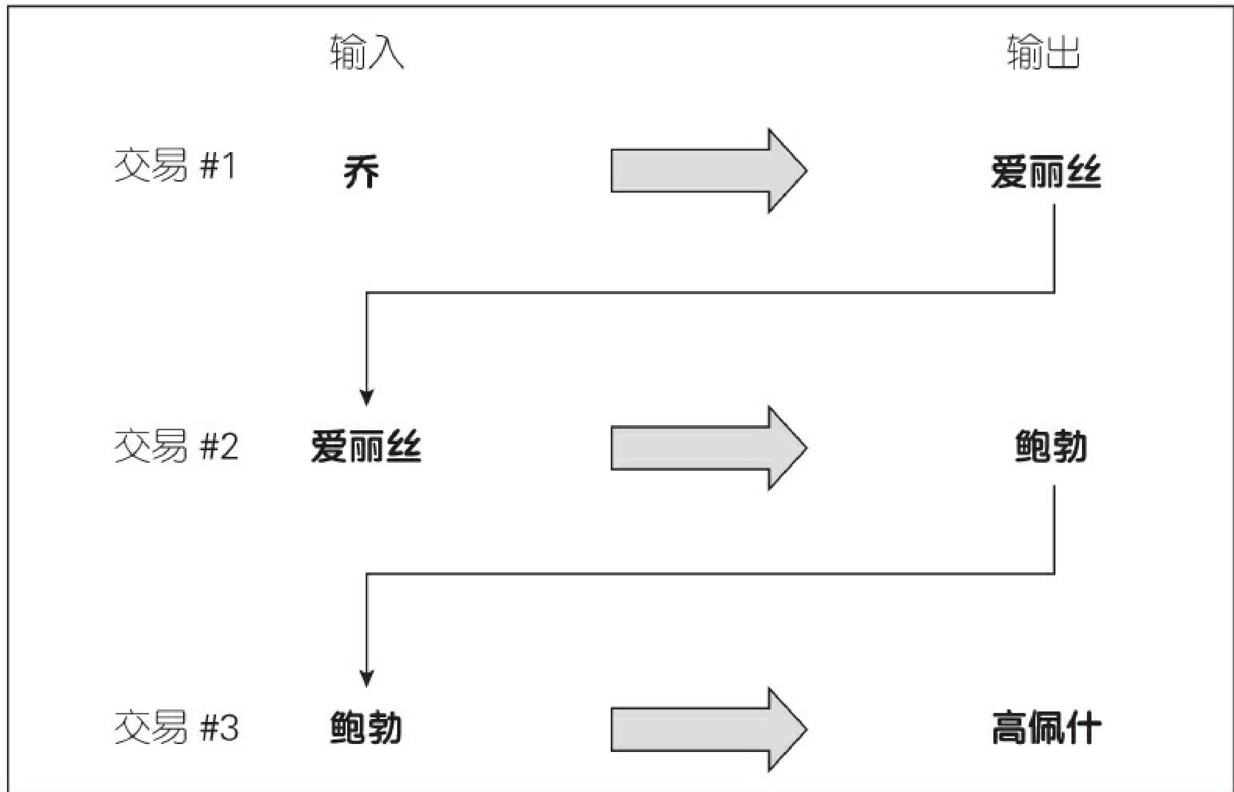


图2.10　爱丽丝的交易成为从乔到高佩什的交易链条的一部分

第3章 比特币客户端

比特币核心：标准客户端

你可以从bitcoin.org下载比特币的标准客户端比特币核心（Bitcoin Core），也被称为“中本聪客户端”。这个标准客户端实现了比特币系统的所有功能，包括钱包、一个交易验证引擎（用于对全部交易账本，即区块链的全量副本进行交易验证），以及一个用于接入点对点比特币网络的完全网络节点。

在“选择你的钱包”页面（**http://bitcoin.org/en/choose-your-wallet**），点击比特币核心，下载标准客户端。基于你的操作系统，你可以下载不同安装包的可执行文件。对于Windows系统，有ZIP压缩包和.exe可执行程序供下载。对于Mac OS，则是一个.dmg磁盘映像。对于Linux版本，包括Ubuntu系统的PPA包或者tar.gz档案。bitcoin.org网页中列出的建议客户端[注]如图3.1所示。



图3.1　从bitcoin.org下载合适的比特币客户端

首次运行比特币核心

如果你已经下载了安装包，比如.exe、.dmg或者PPA，你可以像安装其他任何软件一样，在你的操作系统上安装比特币核心。Windows用户运行.exe，并根据提示一步步进行安装。对于Mac　OS用户，先运行.dmg，完成后将Bitcoin-QT图标拉到应用程序文件夹即可。对于

Ubuntu，在文件浏览器中双击PPA，系统将会打开软件包管理器进行软件包的安装。一旦安装完成，你将在应用程序列表中看到一个新的叫作Bitcoin-QT的软件。双击图标可以启动比特币客户端。

首次运行比特币核心，它将开始下载区块链，这个过程可能需要持续几天（见图3.2）。让它在后台运行，直到显示“同步完成”，并且余额旁不再显示“未同步”。

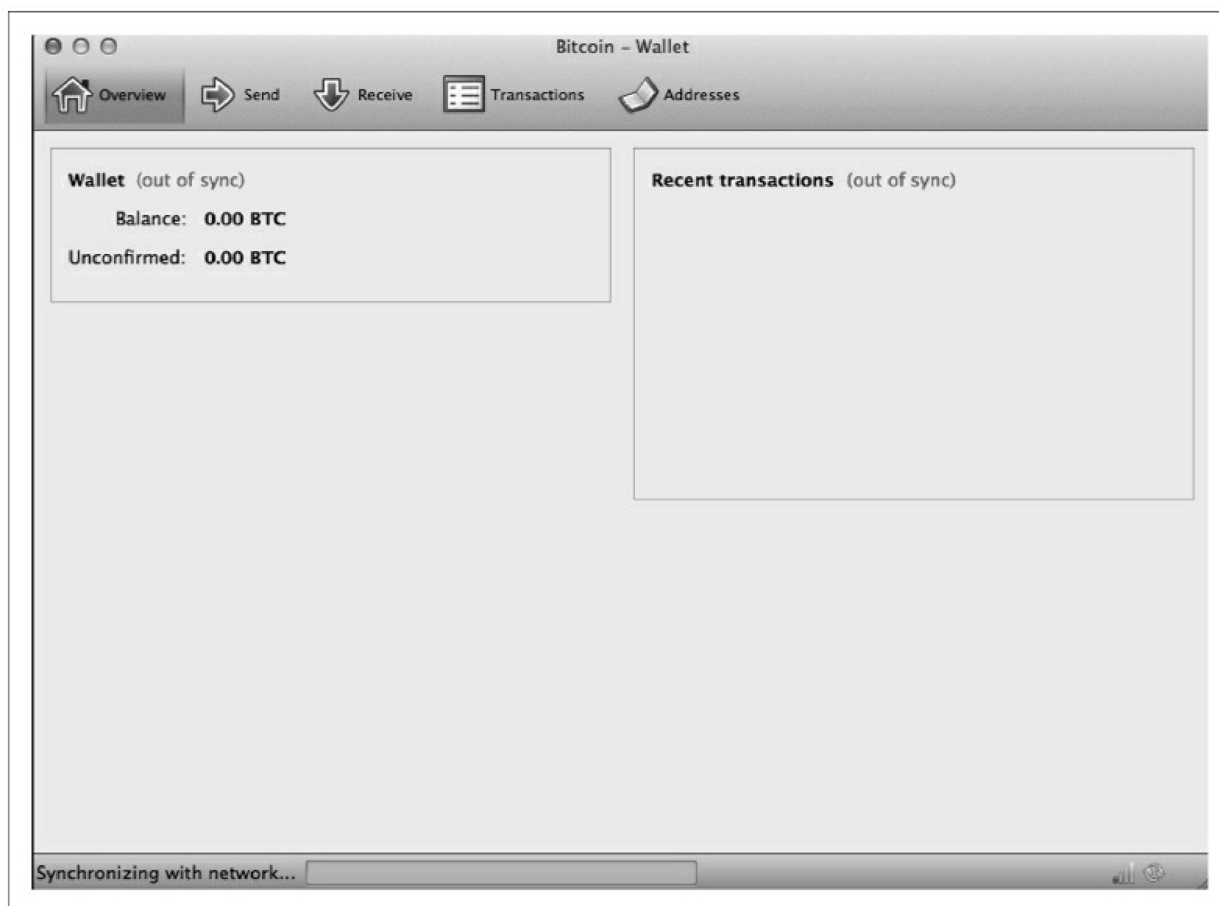


图3.2　比特币核心在区块链初始化时的屏幕显示

比特币核心在本地保存一份交易账本（区块链）的全量副本，包含了自2009年比特币创立以来在比特币网络上发生过的所有交易。数据集大小有几千兆字节（在2013年末大概是16GB），它会以增量的形式，在几天内逐步下载完成。在区块链数据集下载完成之前，客户端都没法执行交易或者更新账户余额。这段时间内，客户端会在账户余额边上显

示“未同步”，下方状态栏则会显示“正在同步”。为了完成初始化同步，请确保你有充足的硬盘空间、网络带宽和足够的时间。

从源代码编译比特币核心

对于开发者来说，也可以选择下载全量的源码（ZIP压缩包）或者从GitHub的官方源码库中复制代码。在GitHub的bitcoin页面（**http://github.com/bitgoin/bitcoin**），从边栏选择下载ZIP包。或者使用git命令行创建一个本地代码库，并从gitHub下载副本。在下面的例子中，我们使用类Unix系统（Linux、Mac　OS等）的命令行，从gitHub上复制代码。

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 31864, done.
remote: Compressing objects: 100% (12007/12007), done.
remote: Total 31864 (delta 24480), reused 26530 (delta 19621)
Receiving objects: 100% (31864/31864), 18.47 MiB | 119 KiB/s, done.
Resolving deltas: 100% (24480/24480), done.
$
```

终端上的提示和输出结果可能会因为版本不同而有所不同。只要按照代码中所带的文档执行，即使实际输出结果与例子中显示的有轻微差异，也是正常的。

当git复制操作完成后，在bitcoin目录中就拥有了一份代码库的完整副本。在提示符下键入命令“cd bitcoin”，进入该目录：

```
$ cd bitcoin
```

不加参数的情况下，检出的本地副本与最新代码保持同步，这可能是比特币的一个不稳定版或公共测试版（beta版）。因此在编译代码

前，应通过加版本标签的形式来检出某个特定版本。这将让本地副本与版本库上某个特定版本的快照进行同步。这些版本标签是利用tag关键词进行标记的，它是开发者使用版本号对特定代码版本进行标记的一种技术。首先，为了找出所有可用标签，我们使用git tag命令：

```
$ git tag
v0.1.5
v0.1.6test1
v0.2.0
v0.2.10
v0.2.11
v0.2.12

[... many more tags ...]

v0.8.4rc2

v0.8.5
v0.8.6
v0.8.6rc1
v0.9.0rc1
```

这个标签列表列出了所有的比特币发行版本。按照惯例，候选发行版（**release candidates**）用于测试目的，带有“rc”后缀。稳定发行版则没有后缀，可以在生产系统上运行。从上述列表中，选择最高版本号的发行版，在写本书时，这个版本是v0.9.0rc1。为了让本地代码与这个版本同步，使用git checkout命令：

```
$ git checkout v0.9.0rc1
Note: checking out 'v0.9.0rc1'.

HEAD is now at 15ec451... Merge pull request #3605
$
```

源码中包含了文档，可以在几个文件中找到。键入more **README.md**，查阅在bitcoin目录中的README.md主文档，根据提示，使用空格键来引导文档翻到下页。在本章中，我们将构建命令行形式的比特币客户端，在linux上又名bitcoind。键入more doc/build-unix.md可以查阅在平台上编译bitcoind命令行客户端的指南。其他平台，如Mac OS X或者Windows的编译指南也可以在**doc**目录下找到，相应的文件为**build-osx.md**或者**build-msw.md**。

仔细研究构建的前置条件，在构建文档的前面部分有描述。这些是在编译bitcoind前必须在系统中安装好的库文件。如果前置条件缺失，构建过程就会失败，并显示错误信息。如果编译过程中发现缺少了某些必需的库文件，你可以在安装好这些库文件后，重新执行编译程序，它将从刚才中断的地方继续进行构建。假设所有前置要求都已经满足，你可以开始利用**autogen.sh**生成一系列构建脚本，开始构建过程。

比特币核心的构建过程从0.9版开始变为采用autogen/configure/make系统。早期版本采用一个简单的Makefile文件，工作过程与下述例子有些细微区别。请按照选定版本的操作指南来操作。0.9版引入的autogen/configure/make构建系统很可能成为所有后续版本的构建方法，也是下面例子演示的构建系统。

```
$ ./autogen.sh
configure.ac:12: installing `src/build-aux/config.guess'
configure.ac:12: installing `src/build-aux/config.sub'
configure.ac:37: installing `src/build-aux/install-sh'
configure.ac:37: installing `src/build-aux/missing'
src/Makefile.am: installing `src/build-aux/depcomp'
$
```

autogen.sh脚本将创建一套自动化配置脚本，这些脚本通过检查你的系统以发现正确的设置，并确保你已经安装了编译代码所需的所有库文件。这些脚本里面最重要的是configure脚本，它提供了一系列不同选

项，帮助你定制构建过程。输入./configure-help，可以查看所有选项：

```
$ ./configure --help

`configure' configures Bitcoin Core 0.9.0 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version           display version information and exit

[... many more options and variables are displayed below ...]

Optional Features:
  --disable-option-checking  ignore unrecognized --enable/--with options
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]

[... more options ...]

Use these variables to override the choices made by `configure' or to help
it to find libraries and programs with nonstandard names/locations.

Report bugs to <info@bitcoin.org>.

$
```

configure脚本允许你通过- -enable-FEATURE或- -disable-FEATURE选项，来启用或禁用bitcoind某些功能。其中的FEATURE由具体功能名称代替，功能名称在上面的帮助信息中已经列出。在本章中，我们构建的bitcoind客户端将打开所有默认功能。我们不配置任何选项，但你最好还是好好研读一下帮助信息，了解客户端包含了哪些可选功能。接下来，运行configure脚本来自动发现所有必要的库，并为系统创建一个定

制的构建脚本。

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes

[... many more system features are tested ...]

configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating src/test/Makefile
config.status: creating src/qt/Makefile
config.status: creating src/qt/test/Makefile
config.status: creating share/setup.nsi
config.status: creating share/qt/Info.plist
config.status: creating qa/pull-tester/run-bitcoind-for-test.sh
config.status: creating qa/pull-tester/build-tests.sh
config.status: creating src/bitcoin-config.h
config.status: executing depfiles commands
$
```

如果一切顺利，configure命令将成功完成定制化构建脚本的创建，这个脚本允许我们编译bitcoind。如果有缺失的库或者其他错误，configure命令将终止创建构建脚本，并输出错误。如果发生错误，很可能是因为缺失库或者库不兼容。再次查阅构建文档，确保已安装了所缺失的先决条件。然后重新运行configure看看是否已修复了错误。接下来，你将编译源代码，这个过程可能会持续一个小时。在编译的过程中，每隔几秒到几分钟就会输出一些信息，如果有什么问题，错误信息

也会显示出来。编译如果被中断，你也可以随时恢复编译过程。键入make开始编译吧。

```
$ make
Making all in src
make[1]: Entering directory `/home/ubuntu/bitcoin/src'
make  all-recursive
make[2]: Entering directory `/home/ubuntu/bitcoin/src'
Making all in .
make[3]: Entering directory `/home/ubuntu/bitcoin/src'
  CXX    addrman.o
  CXX    alert.o
  CXX    rpcserver.o
  CXX    bloom.o
  CXX    chainparams.o

[... many more compilation messages follow ...]

  CXX    test_bitcoin-wallet_tests.o
  CXX    test_bitcoin-rpc_wallet_tests.o
  CXXLD  test_bitcoin
make[4]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[3]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[2]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Entering directory `/home/ubuntu/bitcoin'
make[1]: Nothing to be done for `all-am'.
make[1]: Leaving directory `/home/ubuntu/bitcoin'
$
```

一切运行顺利的话，bitcoind就编译好了。最后的步骤是将bitcoind安装到系统路径中，仍然使用make命令：

```
$ sudo make install
Making install in src
Making install in .
 /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c bitcoind bitcoin-cli '/usr/local/bin'
Making install in test
make  install-am
 /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c test_bitcoin '/usr/local/bin'
$
```

你可以通过查看系统中以下两个可执行程序的位置，来确认bitcoind是否已经安装正确：

```
$ which bitcoind
/usr/local/bin/bitcoind

$ which bitcoin-cli
/usr/local/bin/bitcoin-cli
```

默认安装时，bitcoind会被安装到**/usr/local/bin**目录下。当你第一次运行bitcoind时，它会提醒你创建一个配置文件，这个配置文件包含访问JSON-RPC接口的高强度密码。键入bitcoind，在终端上运行bitcoind：

```
$ bitcoind
Error: To use the "-server" option, you must set a rpcpassword in the configura-
tion file:
/home/ubuntu/.bitcoin/bitcoin.conf
It is recommended you use the following random password:
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK
(you do not need to remember this password)
The username and password MUST NOT be the same.
If the file does not exist, create it with owner-readable-only file permissions.
It is also recommended to set alertnotify so you are notified of problems;
for example: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com
```

在你喜欢的编辑器中编辑配置文件，设置参数，将密码替换为一个bitcoind建议的高强度密码。不要使用范例中使用的密码。在**.bitcoin**目录下创建一个命名为**.bitcoin/bitcoin.conf**的配置文件，输入用户名和密码：

```
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK
```

当你编辑这个配置文件时，你可能还希望设置其他几个选项，比如txindex（参见本章“探索及解码交易”中的附表“交易数据库索引和txindex选项”）。若需要查看所有可用选项，请键入bitcoind- -help。

现在，运行比特币核心客户端。第一次运行时，它会通过下载所有区块来重建比特币区块链。这是一个好几千兆字节的大文件，平均需要花费两天才能全量下载完成。你可以利用BitTorrent客户端从SourceForge（**http://bit.ly/IqkLNyh**）下载部分区块链副本，以此来缩短区块链的初始化时间。

通过-daemon选项可以在后台运行bitcoind：

```
$ bitcoind -daemon

Bitcoin version v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
Using OpenSSL version OpenSSL 1.0.1c 10 May 2012
Default data directory /home/bitcoin/.bitcoin
Using data directory /bitcoin/
Using at most 4 connections (1024 file descriptors available)
init message: Verifying wallet...
dbenv.open LogDir=/bitcoin/database ErrorFile=/bitcoin/db.log
Bound to [::]:8333
Bound to 0.0.0.0:8333
init message: Loading block index...
Opening LevelDB in /bitcoin/blocks/index
Opened LevelDB successfully
Opening LevelDB in /bitcoin/chainstate
Opened LevelDB successfully

[... more startup messages ...]
```

1. 所选截图为翻译此书时的最新客户端列表。——译者注

通过命令行调用比特币核心的**JSON-RPC**接口

比特币核心客户端实现了一个可供命令行助手bitcoin-cli调用的JSON-RPC接口。这使我们可以实验那些通常由程序通过API调用的功能。开始前，我们先调用help命令来看一下全部可用的RPC命令列表：

```
$ bitcoin-cli help
addmultisigaddress nrequired ["key",...] ( "account" )
addnode "node" "add|remove|onetry"
backupwallet "destination"
createmultisig nrequired ["key",...]
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}
decoderawtransaction "hexstring"
decodescript "hex"
dumpprivkey "bitcoinaddress"
dumpwallet "filename"
getaccount "bitcoinaddress"
getaccountaddress "account"
getaddednodeinfo dns ( "node" )
getaddressesbyaccount "account"
getbalance ( "account" minconf )
getbestblockhash
getblock "hash" ( verbose )
getblockchaininfo
getblockcount
getblockhash index
getblocktemplate ( "jsonrequestobject" )
getconnectioncount
getdifficulty
getgenerate
gethashespersec
getinfo
getmininginfo
getnettotals
getnetworkhashps ( blocks height )
getnetworkinfo
getnewaddress ( "account" )
getpeerinfo
getrawchangeaddress
getrawmempool ( verbose )
getrawtransaction "txid" ( verbose )
getreceivedbyaccount "account" ( minconf )
getreceivedbyaddress "bitcoinaddress" ( minconf )
gettransaction "txid"
gettxout "txid" n ( includemempool )
gettxoutsetinfo
getunconfirmedbalance
getwalletinfo
getwork ( "data" )
help ( "command" )
importprivkey "bitcoinprivkey" ( "label" rescan )
```

```
importwallet "filename"
keypoolrefill ( newsize )
listaccounts ( minconf )
listaddressgroupings
listlockunspent
listreceivedbyaccount ( minconf includeempty )
listreceivedbyaddress ( minconf includeempty )
listsinceblock ( "blockhash" target-confirmations )
listtransactions ( "account" count from )
listunspent ( minconf maxconf  ["address",...] )
lockunspent unlock [{"txid":"txid","vout":n},...]
move "fromaccount" "toaccount" amount ( minconf "comment" )
ping
sendfrom "fromaccount" "tobitcoinaddress" amount ( minconf "comment" "comment-
to" )
sendmany "fromaccount" {"address":amount,...} ( minconf "comment" )
sendrawtransaction "hexstring" ( allowhighfees )
sendtoaddress "bitcoinaddress" amount ( "comment" "comment-to" )
setaccount "bitcoinaddress" "account"
setgenerate generate ( genproclimit )
settxfee amount
signmessage "bitcoinaddress" "message"
signrawtransaction "hexstring" ( [{"txid":"id","vout":n,"scriptPub-
Key":"hex","redeemScript":"hex"},...] ["privatekey1",...] sighashtype )
stop
submitblock "hexdata" ( "jsonparametersobject" )
validateaddress "bitcoinaddress"
verifychain ( checklevel numblocks )
verifymessage "bitcoinaddress" "signature" "message"
walletlock
walletpassphrase "passphrase" timeout
walletpassphrasechange "oldpassphrase" "newpassphrase"
```

从比特币核心客户端的状态中获取消息

命令：**getinfo**

比特币的getinfo RPC命令显示比特币网络节点、钱包、区块链数据库状态的基础信息，运行bitcoin-cli：

```
$ bitcoin-cli getinfo
{
    "version" : 90000,
    "protocolversion" : 70002,
    "walletversion" : 60000,
    "balance" : 0.00000000,
    "blocks" : 286216,

    "timeoffset" : -72,
    "connections" : 4,
    "proxy" : "",
    "difficulty" : 2621404453.06461525,
    "testnet" : false,
    "keypoololdest" : 1374553827,
    "keypoolsize" : 101,
    "paytxfee" : 0.00000000,
    "errors" : ""
}
```

数据以JavaScript对象符号（JSON）格式返回，这种格式不仅容易

被编程语言“消费”，也便于人工阅读。在数据中，我们可以看到比特币客户端的版本号（90000），协议版本号（70002），钱包版本号（60000），也能看到钱包的余额，当前为0。还能看到区块高度，告诉我们客户端总共看到了多少区块（当前286216）。另外，返回信息中还包含各种比特币网络相关的统计数据，以及当前客户端相关的设置信息。我们将在本章剩余部分更详细地了解这些设置。

这将需要一些时间，也许超过一天，等待bitcoind客户端从其他比特币客户端下载区块，以“赶上”当前的区块链高度。你可以通过getinfo查看已知的区块来检查同步进度。

钱包设置和加密

命令：**encryptwallet, walletpassphrase**

在开始创建密钥和其他命令前，你需要首先利用密码来给钱包加密。在这个例子中，我们使用encryptwallet命令和密码“foo”，显然用更加强大和复杂的密码替代“foo”是必须的！

```
$ bitcoin-cli encryptwallet foo
wallet encrypted; Bitcoin server stopping, restart to run with encrypted wal-
let. The keypool has been flushed, you need to make a new backup.
$
```

你可以重新运行getinfo来检查钱包是否已经加密。这次，你会注意到有个新的条目叫unlocked_until。这是一个计数器，显示钱包解密密码在内存中存储、保持钱包解锁状态的时间。最初，计数器会被设为0，代表钱包是被锁定的：

```
$ bitcoin-cli getinfo
{
    "version" : 90000,

#[... other information...]
    "unlocked_until" : 0,
    "errors" : ""
}
$
```

为了解锁钱包，发出walletpassphrase命令，它包含两个参数——密码、直到钱包重新锁定的超时秒数（时间计数器）：

```
$ bitcoin-cli walletpassphrase foo 360
$
```

你可以重新运行getinfo确认钱包是否已解锁并获取超时时间：

```
$ bitcoin-cli getinfo
{
    "version" : 90000,

#[... other information ...]

    "unlocked_until" : 1392580909,
    "errors" : ""
}
```

钱包备份，明文导出，恢复

命令：**backupwallet, importwallet, dumpwallet**

接下来，我们练习创建钱包备份文件，然后从备份文件中恢复钱包。使用backupwallet命令来备份，提供文件名作为命令的参数。这里，我们将钱包备份到文件**wallet.backup**中：

```
$ bitcoin-cli backupwallet wallet.backup
$
```

现在，利用importwallet从备份文件中恢复钱包。如果钱包是锁定状态的，你需要先进行解锁（在上一节可以查看walletpassphrase命令用

法）。

```
$ bitcoin-cli importwallet wallet.backup
$
```

dumpwallet命令用于将钱包导出为可读的文本文件：

```
$ bitcoin-cli dumpwallet wallet.txt
$ more wallet.txt
# Wallet dump created by Bitcoin v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
# * Created on 2014-02- 8dT20:34:55Z
# * Best block at time of backup was 286234
(0000000000000000f74f0bc9d3c186267bc45c7b91c49a0386538ac24c0d3a44),
#   mined on 2014-02- 8dT20:24:01Z

KzTg2wn6Z8s7ai5NA9MVX4vstHRsqP26QKJCzLg4JvFrp6mMaGB9 2013-07- 4dT04:30:27Z
change=1 # addr=16pJ6XkwSQv5ma5FSXMRPaXEYrENCEg47F
Kz3dVz7R6mUpXzdZy4gJEVZxXJwA15f198eVui4CUivXotzLBDKY 2013-07- 4dT04:30:27Z
change=1 # addr=17oJds8kaN8LP8kuAkWTco6ZM7BGXFC3gk
[... many more keys ...]

$
```

钱包地址和接收交易

命令：**getnewaddress, getreceivedbyaddress, listtransactions, getaddressesbyaccount, getbalance**

比特币标准客户端维护着一个地址池，地址池的大小在命令getinfo的输出项keypoolsize中展示。这些地址是自动生成的，可作为公开的接收地址或者找零地址。使用getnewaddress命令，可以生成一个新地址：

```
$ bitcoin-cli getnewaddress
1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
```

现在，我们可以从外部钱包（假设你在交易所、网络钱包或者在其他地方运行的bitcoind钱包中已经拥有一些比特币），通过这个地址发送一笔小额的比特币到我们的bitcoind钱包。在这个例子中，我们将发送50毫比特（0.050比特币）到这个地址。

我们可以通过bitcoind客户端查询此地址已经接收到的比特币金额。查询时，需要指定确认次数，即经过多少次确认一笔资金才会计入余额中，在这里，我们指定0次确认。从另一个钱包发送比特币几秒后，我们就可以看到，交易已经在钱包的余额中体现了。现在，我们使用getreceivedbyaddress命令并结合地址及确认次数0来查看：

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL 0
0.05000000
```

如果省略掉命令最后一个0，我们将只能看到经过至少minconf次确认的金额，minconf是最少确认次数的设置值，未达到这个确认次数，交易不会计入余额。minconf的值是在bitcoind配置文件中设置的。因为发送这个比特币的交易仅仅发生在几秒前，它尚未被确认，所以我们看到它只列出了一个0余额：

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
0.00000000
```

钱包接收到的所有交易也可以利用listtransactions命令列出来：

```
$ bitcoin-cli listtransactions

[
    {
        "account" : "",
        "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
        "category" : "receive",
        "amount" : 0.05000000,
```

```
        "confirmations" : 0,
        "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309ac
bae2c14ae3",
        "time" : 1392660908,
        "timereceived" : 1392660908
    }
]
```

我们还可以利用getaddressesbyaccount命令列出钱包中的所有地址：

```
$ bitcoin-cli getaddressesbyaccount ""

[
    "1LQoTPYy1TyERbNV4zZbhEmgyfAipC6eqL",
    "17vrg8uwMQUibkvS2ECRX4zpcVJ78iFaZS",
    "1FvRHWhHBBZA8cGRRsGiAeqEzUmjJkJQWR",
    "1NVJK3JsL41BF1KyxrUyJW5XHjunjfp2jz",
    "14MZqqzCxjc99M5ipsQSRfieT7qPZcM7Df",
    "1BhrGvtKFjTAhGdPGbrEwP3xvFjkJBuFCa",
    "15nem8CX91XtQE8B1Hdv97jE8X44H3DQMT",
    "1Q3q6taTsUiv3mMemEuQQJ9sGLEGaSjo81",
    "1HoSiTg8sb16oE6SrmazQEwcGEv8obv9ns",
    "13fE8BGhBvnoy68yZKuWJ2hheYKovSDjqM",
    "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
    "1KHUmVfCJteJ21LmRXHSpPoe23rXKifAb2",
    "1LqJZz1D9yHxG4cLkdujnqG5jNNGmPeAMD"
]
```

最后，命令getbalance可以显示钱包的全部余额，命令会自动合并所有经过至少minconf次确认的交易金额。

```
$ bitcoin-cli getbalance
0.05000000
```

如果交易尚未确认，getbalance命令返回的余额是0。“minconf”选项决定了需要经过几次确认，交易金额才会体现到余额上。

探索及解码交易

命令：**gettransaction**，**getrawtransaction**，**decoderawtransaction**

现在，我们来看看前面利用gettransaction命令列出的传入交易。我们可以通过交易哈希提取到一笔交易，交易哈希就是前面我们看到的txid，提取交易的命令为：gettransaction。

```
$ bitcoin-cli gettransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66↵
c309acbae2c14ae3
```

```
{
    "amount" : 0.05000000,
    "confirmations" : 0,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908,
    "details" : [
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.05000000
        }
    ]
}
```

在未确认前，交易ID不具有权威性。区块链中交易哈希缺失不意味着交易未被执行，这被称为“交易的可塑性”，因为交易哈希可以在区块确认前被修改。一旦确认，txid就是不变的，权威的。

通过命令gettransaction显示的交易形式是一种简化的形式。为了获取完整交易的代码并解码它，我们需要利用两个命令：getrawtransaction和decoderawtransaction。首先，使用命令getrawtransaction，并以交易哈希（**txid**）作为参数，将返回一个“原始”的十六进制字符串，就像它在比特币网络上的样子。

```
$ bitcoin-cli getrawtransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1f↵
c66c309acbae2c14ae30100000001d717279515f88e2f56ce4e8a31e2ae3e9f00ba1d0add648e↵
80c480ea22e0c7d3000000008b483045022100a4ebbeec83225dedead659bbde7da3d026c8b8e↵
12e61a2df0dd0758e227383b302203301768ef878007e9ef7c304f70ffaf1f2c975b192d34c5b↵
9b2ac1bd193dfba2014104793ac8a58ea751f9710e39aad2e296cc14daa44fa59248be58ede65↵
e4c4b884ac5b5b6dede05ba84727e34c8fd3ee1d6929d7a44b6e111d41cc79e05dbfe5ceaffff↵
ffff02404b4c00000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac1f3↵
1290600000000001976a914107b7086b31518935c8d28703d66d09b3623134388ac00000000
```

为了解码这个十六进制字符串，需要使用decoderawtransaction命令。复制粘贴这些十六进制代码作为命令decoderawtransaction的第一个

参数，就得到了完整的以JSON数据格式表示的内容（出于排版格式需要，十六进制字符串在以下例子中被截短了）：

```
$ bitcoin-cli decoderawtransaction 0100000001d717…388ac00000000
{
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "d3c7e022ea80c4808e64dd0a1dba009f3eaee2318a4ece562f8ef815↵
952717d7",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "3045022100a4ebbeec83225dedead659bbde7da3d026c8b8e12e↵
61a2df0dd0758e227383b302203301768ef878007e9ef7c304f70ffaf1f2c975b192d34c5b9b2↵
ac1bd193dfba20104793ac8a58ea751f9710e39aad2e296cc14daa44fa59248be58ede65e4c4b↵
884ac5b5b6dede05ba84727e34c8fd3ee1d6929d7a44b6e111d41cc79e05dbfe5cea",
                "hex": "483045022100a4ebbeec83225dedead659bbde7da3d026c8b8e1↵
2e61a2df0dd0758e227383b302203301768ef878007e9ef7c304f70ffaf1f2c975b192d34c5b9↵
b2ac1bd193dfba2014104793ac8a58ea751f9710e39aad2e296cc14daa44fa59248be58ede65e↵
4c4b884ac5b5b6dede05ba84727e34c8fd3ee1d6929d7a44b6e111d41cc79e05dbfe5cea"
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.05000000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c↵
13d1fd2 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"
                ]
            }
```

```
        },
        {
            "value" : 1.03362847,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 107b7086b31518935c8d28703d66d09b36↵
231343 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a914107b7086b31518935c8d28703d66d09b3623134388ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "12W9goQ3P7Waw5JH8fRVs1e2rVAKoGnvoy"
                ]
            }
        }
    ]
}
```

这个JSON格式的输出展示了一个交易的所有组成元素，包括交易输入和输出。在这个例子中，我们看到交易使用了一个输入并生成了两个输出，在我们的新地址记入了50毫比特。这个交易的输入是上一笔已确认交易的输出（显示为vin下d3c7开始的txid）。两个输出一笔是50毫比特的入账，另一笔是交易找零。

我们可以使用同样的命令（如gettransaction），通过查看txid引用的交易，进一步对区块链进行探索。如此一级一级循着交易链条，我们可以看到资金一次又一次地从一个所有者地址转移到另一个所有者地址。

如果接收到的交易已经被确认，gettransaction命令还将额外返回交易所在区块的区块哈希（标识符）：

```
$ bitcoin-cli gettransaction 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66↵
c309acbae2c14ae3
{
    "amount" : 0.05000000,
    "confirmations" : 1,
    "blockhash" : "000000000000000051d2e759c63a26e247f185ecb7926ed7a6624bc31c↵
2a717b",
    "blockindex" : 18,
    "blocktime" : 1392660808,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908,
    "details" : [
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.05000000
        }
    ]
}
```

在这里，我们看到输出项中多了一个blockhash条目（区块哈希，交易所在区块的哈希值）和值为18的blockindex条目（区块索引，指此交易是区块内的第18个交易）

交易数据库索引和txindex选项

默认情况下，比特币核心创建一个仅包含与用户钱包相关的交易数据库。如果你希望访问任何类似gettransaction等命令能查看的交易，你需要配置比特币核心，使其创建一个完整的交易索引。这可以通过设置txindex选项来实现。在比特币核心的配置文件（该文件通常位于你的Home目录下的.bitcoin/ bitcoin.conf）中设置txindex=1。一旦你修改了参数，你需要重启bitcoind并等待其完成索引创建。

探索区块

命令：**getblock, getblockhash**

现在我们已经知道交易位于哪个区块内，我们可以查询该区块。使用getblock命令并以区块哈希作为参数：

```
$ bitcoin-cli getblock 000000000000000051d2e759c63a26e247f185ecb7926ed7a6624b↵
c31c2a717b true

{
    "hash" : "000000000000000051d2e759c63a26e247f185ecb7926ed7a6624bc31c2a717↵
b",
    "confirmations" : 2,
    "size" : 248758,
    "height" : 286384,
```

```
    "version" : 2,
    "merkleroot" : "9891747e37903016c3b77c7a0ef10acf467c530de52d84735bd555387↵
19f9916",
    "tx" : [
        "46e130ab3c67d31d2b2c7f8fbc1ca71604a72e6bc504c8a35f777286c6d89bf0",
        "2d5625725b66d6c1da88b80b41e8c07dc5179ae2553361c96b14bcf1ce2c3868",
        "923392fc41904894f32d7c127059bed27dbb3cfd550d87b9a2dc03824f249c80",
        "f983739510a0f75837a82bfd9c96cd72090b15fa3928efb9cce95f6884203214",
        "190e1b010d5a53161aa0733b953eb29ef1074070658aaa656f933ded1a177952",
        "ee791ec8161440262f6e9144d5702f0057cef7e5767bc043879b7c2ff3ff5277",
        "4c45449ff56582664abfadeb1907756d9bc90601d32387d9cfd4f1ef813b46be",
        "3b031ed886c6d5220b3e3a28e3261727f3b4f0b29de5f93bc2de3e97938a8a53",
        "14b533283751e34a8065952fd1cd2c954e3d37aaa69d4b183ac6483481e5497d",
        "57b28365adaff61aaf60462e917a7cc9931904258127685c18f136eeaebd5d35",
        "8c0cc19fff6b66980f90af39bee20294bc745baf32cd83199aa83a1f0cd6ca51",
        "1b408640d54a1409d66ddaf3915a9dc2e8a6227439e8d91d2f74e704ba1cdae2",
        "0568f4fad1fdeff4dc70b106b0f0ec7827642c05fe5d2295b9deba4f5c5f5168",
        "9194bfe5756c7ec04743341a3605da285752685b9c7eebb594c6ed9ec9145f86",
        "765038fc1d444c5d5db9163ba1cc74bba2b4f87dd87985342813bd24021b6faf",
        "bff1caa9c20fa4eef33877765ee0a7d599fd1962417871ca63a2486476637136",
        "d76aa89083f56fcce4d5bf7fcf20c0406abdac0375a2d3c62007f64aa80bed74",
        "e57a4c70f91c8d9ba0ff0a55987ea578affb92daaa59c76820125f31a9584dfc",
        "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",

#[... many more transactions ...]

    ],
    "time" : 1392660808,
    "nonce" : 3888130470,
    "bits" : "19015f53",
    "difficulty" : 3129573174.52228737,
    "chainwork" : "000000000000000000000000000000000000000000001931d1658fc048↵
79e466",
    "previousblockhash" : "0000000000000000177e61d5f6ba6b9450e0dade9f39c257b4↵
d48b4941ac77e7",
    "nextblockhash" : "0000000000000001239d2c3bf7f4c68a4ca673e434702a57da8fe0↵
d829a92eb6"
```

区块包含367个交易，你可以看到，第18个交易（9ca8f9...）是往我们的地址上发送50毫比特的交易ID（txid）。输出项高度（height）告诉我们这是区块链中的第286384个区块。

我们也可以利用getblockhash命令，根据区块的高度来获取区块信息。这种情况下，我们使用区块高度作为参数，返回区块哈希：

```
$ bitcoin-cli getblockhash 000000000019d6689c085ae165831e934ff763ae46a2a6c17↵
2b3f1b60a8ce26f
```

这里，我们取得了“创世区块”的哈希，它是中本聪挖到的第一个区块，其高度为0。进一步获取区块的详细信息如下：

```
$ bitcoin-cli getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1↵
b60a8ce26f

{
    "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26↵
f",
    "confirmations" : 286388,
    "size" : 285,
    "height" : 0,
    "version" : 1,
    "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7af↵
deda33b",
    "tx" : [
        "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
    ],
    "time" : 1231006505,
    "nonce" : 2083236893,
    "bits" : "1d00ffff",
    "difficulty" : 1.00000000,
    "chainwork" : "000000000000000000000000000000000000000000000000000000000100↵
010001",
    "nextblockhash" : "00000000839a8e6886ab5951d76f411475428afc90947ee320161b↵
bf18eb6048"
}
```

getblock、getblockhash，以及gettransaction命令也可用于程序中查询区块链数据库。

基于未花费输出的创建、签名、提交交易

命令：**listunspent, gettxout, createrawtransaction,**

decoderawtransaction, signrawtransaction, sendrawtransaction

比特币的交易基于花费“输出”的概念，“输出”是前序交易的结果，由此创建了一个在所有者地址间的价值传递的交易链。我们的钱包软件现在接收到了一个交易，它将一个输出分配给我们的地址。一旦这个交易被确认，我们就能花费这个输出。

首先：我们利用listunspent命令来显示我们钱包中未花费的已确认交易输出：

```
$ bitcoin-cli listunspent

[
    {
        "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c↵
14ae3",
        "vout" : 0,
        "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
        "account" : "",
        "scriptPubKey" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
        "amount" : 0.05000000,
        "confirmations" : 7
    }
]
```

我们看到交易9ca8f9...创建了一个输出（vout索引号为0），为地址1hvzSo...存入50毫比特，此时该交易已经经过7次确认。交易使用前序交易创建的输出作为它的输入，通过引用前序交易的txid和vout索引形成与前序交易的连接。我们现在可以创建一个新的交易来花费txid为9ca8f9...的第0个输出（vout=0），在这个交易中，我们将把上述交易的输出作为新交易的输入，并将它发给一个新的地址。

首先，我们更细致地了解一下这个交易的输出。我们使用gettxout来获取这个未花费输出。交易输出总是通过txid和vout被引用，这也是我们传给gettxout的参数：

```
$ bitcoin-cli gettxout 9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309ac↵
bae2c14ae3 0
{
    "bestblock" : "0000000000000001405ce69bd4ceebcdfdb537749cebe89d371eb37e13↵
899fd9",
    "confirmations" : 7,
    "value" : 0.05000000,
    "scriptPubKey" : {
        "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c13d1fd2
         OP_EQUALVERIFY OP_CHECKSIG",
        "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
        "reqSigs" : 1,
        "type" : "pubkeyhash",
        "addresses" : [
            "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"
        ]
    },
    "version" : 1,
    "coinbase" : false
}
```

我们在此看到的是给我们的地址1hvz...发送50毫比特的交易输出。为了花费这个输出，我们将创建一笔新交易。首先，我们生成一个新的地址用于接收资金：

```
$ bitcoin-cli getnewaddress
1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb
```

我们将发送25毫比特到这个新创建的地址（1LnfTn...）。在新交易中，我们将花费50毫比特的输出，并且发送25毫比特到这个新地址。由于我们必须花费前序交易的整个输出，所以交易还会产生一部分找零。我们将找零发送回原地址（1hvz...）。最后，我们还需要支付一些交易费用。为了发送费用，我们把找零的数额减少0.5毫比特，仅找回24.5毫比特。交易输出的总额为（25mBTC+24.5mBTC=49.5mBTC）与输入（50mBTC）的差额，将会作为交易费用被矿工收集走。

我们使用createrawtransaction来创建这个交易。作为参数，我们提

供交易输入（已确认交易的50毫比特未花费输出），以及两个交易输出（发往新地址的资金和返回发送者自身地址的交易找零）：

```
$ bitcoin-cli createrawtransaction '[{"txid" : "9ca8f969bd3ef5ec2a8685660fdbf↵
7a8bd365524c2e1fc66c309acbae2c14ae3", "vout" : 0}]' '{"1LnfTndy3qzXGN19Jwscj1↵
T8LR3MVe3JDb": 0.025, "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL": 0.0245}'

0100000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53ebd69f9a89c000↵
0000000ffffffff02a0252600000000001976a914d90d36e98f62968d2bc9bbd68107564a156a↵
9bcf88ac50622500000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac0↵
0000000
```

createrawtransaction命令产生一串原始十六进制字符串，将我们提供的交易细节进行编码。接下来，我们使用decoderawtransaction命令解码这串字符串，来确认一下所有事情是不是已经正确完成。

```
$ bitcoin-cli decoderawtransaction 0100000001e34ac1e2baac09c366fce1c2245536bd↵
a8f7db0f6685862aecf53ebd69f9a89c0000000000ffffffff02a0252600000000001976a914d↵
90d36e98f62968d2bc9bbd68107564a156a9bcf88ac50622500000000001976a91407bdb518fa↵
2e6089fd810235cf1100c9c13d1fd288ac00000000
```

```
{
    "txid" : "0793299cb26246a8d24e468ec285a9520a1c30fcb5b6125a102e3fc05d4f3cb↵
a",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acb↵
ae2c14ae3",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "",
                "hex" : ""
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.02500000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 d90d36e98f62968d2bc9bbd68107564a15↵
6a9bcf OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb"
                ]
            }
        },
        {
            "value" : 0.02450000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c1↵
3d1fd2 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"
                ]
            }
        }
    ]
}
```

看起来像是正确的！新交易“花费”了我们已确认交易中的未花费输出，并将其拆分为两个输出：一个是25毫比特，发到我们新地址；另一个24.5毫比特作为交易找零返回原地址。0.5毫比特的差额作为交易费用，将发给找到包含本交易区块的矿工。

就像你可能注意到的那样，交易中包含一个空的scriptSig（脚本签名），因为我们尚未对其签名。如果没有签名，那么一笔交易是毫无意义的，因为尚未证明我们拥有未花费输出的地址。通过签名，我们移除了输出的限制，证明了我们确实拥有这笔输出，并且能够使用它。我们使用signrawtransaction命令来签名交易，它以原始交易的十六进制字符串作为参数。

一个加密的钱包软件必须先进行解锁，才能对交易进行签名，因为签名需要访问钱包中的密钥。

```
$ bitcoin-cli walletpassphrase foo 360
$ bitcoin-cli signrawtransaction 0100000001e34ac1e2baac09c366fce1c2245536bda8↵
f7db0f6685862aecf53ebd69f9a89c0000000000ffffffff02a0252600000000001976a914d90↵
d36e98f62968d2bc9bbd68107564a156a9bcf88ac50622500000000001976a91407bdb518fa2e↵
6089fd810235cf1100c9c13d1fd288ac00000000

{
    "hex" : "0100000001e34ac1e2baac09c366fce1c2245536bda8f7db0f6685862aecf53e↵
bd69f9a89c000000006a47304402203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86a54↵
e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e4af↵
52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc512↵
7ffffffff02a0252600000000001976a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88↵
ac50622500000000001976a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac00000000",
    "complete" : true
}
```

signrawtransaction命令将另外一个十六进制编码的原始交易返回。我们将其解码之后看看有什么变化：

```
$ bitcoin-cli decoderawtransaction 0100000001e34ac1e2baac09c366fce1c2245536bda↵
8f7db0f6685862aecf53ebd69f9a89c000000006a47304402203e8a16522da80cef66bacfbc0c↵
800c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87↵
e3897a273ef54b277e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac↵
1d3f00fd44ea6c71dc5127ffffffff02a0252600000000001976a914d90d36e98f62968d2bc9b↵
bd68107564a156a9bcf88ac50622500000000001976a91407bdb518fa2e6089fd810235cf1100↵
c9c13d1fd288ac00000000
```

```
{
    "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b34↵
6",
```

```
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acb↵
ae2c14ae3",
            "vout" : 0,
            "scriptSig" : {
                "asm" : "304402203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d86↵
a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b277e↵
4af52051a0601 03c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71dc5↵
127",
                "hex" : "47304402203e8a16522da80cef66bacfbc0c800c6d52c4a26d1d↵
86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e3897a273ef54b27↵
7e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1d3f00fd44ea6c71↵
dc5127"
            },
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 0.02500000,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 d90d36e98f62968d2bc9bbd68107564a15↵
6a9bcf OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a914d90d36e98f62968d2bc9bbd68107564a156a9bcf88ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb"
                ]
            }
        },
        {
            "value" : 0.02450000,
            "n" : 1,
            "scriptPubKey" : {
                "asm" : "OP_DUP OP_HASH160 07bdb518fa2e6089fd810235cf1100c9c1↵
3d1fd2 OP_EQUALVERIFY OP_CHECKSIG",
                "hex" : "76a91407bdb518fa2e6089fd810235cf1100c9c13d1fd288ac",
                "reqSigs" : 1,
                "type" : "pubkeyhash",
                "addresses" : [
                    "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL"
                ]
            }
        }
    ]
}
```

现在，交易中的输入包含了scriptSig，这是一个地址（1hvz...）拥有者提供的数字签名，解除了原交易输出的限制，输出得以使用。签名使交易可以被比特币网络上的任何节点进行确认。

是时候将这个新交易提交到网络中去了。我们使用sendrawtransaction命令，该命令以上述已签名交易的原始十六进制字符串作为参数，也就是刚刚我们解码的字符串。

```
$ bitcoin-cli sendrawtransaction 0100000001e34ac1e2baac09c366fce1c2245536bda8↵
f7db0f6685862aecf53ebd69f9a89c000000006a47304402203e8a16522da80cef66bacfbc0c8↵
00c6d52c4a26d1d86a54e0a1b76d661f020c9022010397f00149f2a8fb2bc5bca52f2d7a7f87e↵
3897a273ef54b277e4af52051a06012103c9700559f690c4a9182faa8bed88ad8a0c563777ac1↵
d3f00fd44ea6c71dc5127ffffffff02a0252600000000001976a914d90d36e98f62968d2bc9bb↵
d68107564a156a9bcf88ac50622500000000001976a91407bdb518fa2e6089fd810235cf1100c↵
9c13d1fd288ac00000000ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584↵
ac17b346
```

当交易提交到网络后，sendrawtransaction命令返回一个交易哈希（txid）。我们可以利用gettransaction命令来查询这个交易ID：

```
$ bitcoin-cli gettransaction ae74538baa914f3799081ba78429d5d84f36a0127438e9f7↵
21dff584ac17b346

{
    "amount" : 0.00000000,
    "fee" : -0.00050000,
    "confirmations" : 0,
    "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",
    "time" : 1392666702,
    "timereceived" : 1392666702,
    "details" : [
        {
            "account" : "",
            "address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
            "category" : "send",
            "amount" : -0.02500000,
            "fee" : -0.00050000
        },
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "send",
            "amount" : -0.02450000,
            "fee" : -0.00050000
        },
        {
            "account" : "",
            "address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
            "category" : "receive",
            "amount" : 0.02500000
        },
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.02450000
        }
    ]
}
```

就像之前看到的，我们依然可以使用getrawtransaction和decodetransaction命令查看更详细的信息。它们返回的信息与交易提交

网络之前看到的是一样的。

替代客户端、库、工具集

除了标准客户端（bitcoind），其他客户端和库也能用来与比特币网络及其数据结构进行交互。这里是一些通过不同语言实现的客户端，提供了相应语言的本地化接口。

libbitcoin和**sx**工具集

一个C++多线程完全客户端和库，包含命令行工具。

bitcoinj

一个Java完全节点客户端库。

btcd

一个Go语言完全节点比特币客户端。

Bits of Proof（BOP）

一个Java实现的企业级比特币客户端。

picocoin

一个轻量级比特币客户端库的C语言实现。

pybitcointools

一个Python实现的比特币库。

pycoin

另一个Python比特币库。

还有很多其他各种语言实现的库，同时还有更多的库正在开发中。

Libbitcoin和sx工具集

libbitcoin库是一个C++的可扩展多线程、模块化的实现，支持完全客户端，它还带一个命令行工具集，叫作sx，这个工具集提供了很多与我们在本章中展示过的bitcoind客户端命令行一样的功能。sx工具集还提供一些bitcoind未提供的密钥管理和维护工具，包括type-2确定性密钥和密钥助记符功能。

安装**sx**

为了安装sx及其支持库libbitcoin，在Linux系统上下载并安装在线安装包。

```
$ wget http://sx.dyne.org/install-sx.sh
$ sudo bash ./install-sx.sh
```

现在，sx工具集已经安装好了，键入不带参数的sx命令打印帮助文档，将会列出所有可用命令（参看附录D）。

sx工具集提供了许多实用命令来对地址进行编码或解码，也可以将它们的格式或者表现方式互相转换。可以利用它们来探索各种不同的格式，比如Base58，Base58Check，十六进制，等等。

pycoin

Python库**pycoin**（**http：//github.com/richardkiss/pycoin**），最早由

理查德·吉斯（Richard Kiss）开发和维护，是一个基于Python的库，它支持操作比特币密钥和交易，甚至也支持使用脚本语言来正确处理非标准交易。

pycoin库支持Python 2（2.7.x）和Python 3（3.3之后版本），同时还附带了一些好用的命令行工具，ku和tx。在Python 3的虚拟环境（venv）安装pycoin 0.42的步骤如下：

```
$ python3 -m venv /tmp/pycoin
$ . /tmp/pycoin/bin/activate
$ pip install pycoin==0.42
Downloading/unpacking pycoin==0.42
  Downloading pycoin-0.42.tar.gz (66kB): 66kB downloaded
  Running setup.py (path:/tmp/pycoin/build/pycoin/setup.py) egg_info for pack-
age pycoin

Installing collected packages: pycoin
  Running setup.py install for pycoin

    Installing tx script to /tmp/pycoin/bin
    Installing cache_tx script to /tmp/pycoin/bin
    Installing bu script to /tmp/pycoin/bin
    Installing fetch_unspent script to /tmp/pycoin/bin
    Installing block script to /tmp/pycoin/bin
    Installing spend script to /tmp/pycoin/bin
    Installing ku script to /tmp/pycoin/bin
    Installing genwallet script to /tmp/pycoin/bin
Successfully installed pycoin
Cleaning up...
$
```

以下是一个利用pycoin库来获取并花费比特币的Python脚本：

```python
#!/usr/bin/env python

from pycoin.key import Key

from pycoin.key.validate import is_address_valid, is_wif_valid
from pycoin.services import spendables_for_address
from pycoin.tx.tx_utils import create_signed_tx

def get_address(which):
    while 1:
        print("enter the %s address=> " % which, end='')
        address = input()
        is_valid = is_address_valid(address)
        if is_valid:
            return address
        print("invalid address, please try again")

src_address = get_address("source")
spendables = spendables_for_address(src_address)
print(spendables)

while 1:
    print("enter the WIF for %s=> " % src_address, end='')
    wif = input()
    is_valid = is_wif_valid(wif)
    if is_valid:
        break
    print("invalid wif, please try again")

key = Key.from_text(wif)
if src_address not in (key.address(use_uncompressed=False), key.address(use_un
compressed=True)):
    print("** WIF doesn't correspond to %s" % src_address)
print("The secret exponent is %d" % key.secret_exponent())

dst_address = get_address("destination")

tx = create_signed_tx(spendables, payables=[dst_address], wifs=[wif])

print("here is the signed output transaction")
print(tx.as_hex())
```

命令行工具ku和tx的例子参看附录B。

btcd

btcd是一个Go语言开发的完全节点比特币客户端。目前，它的下载、验证、服务区块链的规则与标准客户端bitcoind接受区块的规则完全一致（甚至连bug也一致）。它同样能够正确中转新挖出的区块，维护一个交易池，转发尚未进入区块的交易。它确保获准进入交易池的交易严格遵守通用规则及大多数矿工提出的更为严格的过滤原则（“标准”交易）。

btcd和bitcoind之间主要的区别之一在于，btcd没有钱包功能，这是btcd故意的设计选择。这意味着，用户无法直接使用btcd发送或接收支付款项。钱包功能由btcwallet和btcgui提供，这两个软件都尚在开发中。其他比较显著的区别还包括：btcd同时支持HTTP POST请求（bitcoind也支持）和默认的Websocket连接，实际上，btcd的RPC连接是默认启用TLS的[注]。

安装**btcd**

要在Windows下安装btcd，需要从GitHub（**http://github.com/conformal/btcd/releases**）上下载msi安装包，并进行安装；如果是Linux系统，假设已安装了Go语言，则命令如下：

```
$ go get github.com/conformal/btcd/...
```

若需要更新btcd版本，可以直接执行：

```
$ go get -u -v github.com/conformal/btcd/...
```

控制**btcd**

btcd有一系列配置选项，你可以通过以下命令查看：

```
$ btcd --help
```

btcd安装时附带了一些好东西，比如btcctl，这是一个命令行工具，可以通过RPC控制或查询btcd。btcd默认没有启用其RPC服务，要启动该服务，你至少需要在文件**btcd.conf**和**btcctl.conf**中配置好RPC的用户名和密码：

- btcd.conf

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

- btcctl.conf

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

或者你也可以通过命令行来覆盖配置文件，命令如下：

```
$ btcd -u myuser -P SomeDecentp4ssw0rd
$ btcctl -u myuser -P SomeDecentp4ssw0rd
```

要获取可用选项的列表，可以运行以下命令：

```
$ btcctl --help
```

1. TLS（安全传输层协议），用于确保两个通信应用程序之间的保密性和数据完整性。——译者注

第4章 密钥、地址、钱包

介绍

比特币的所有权建立在数字密钥、比特币地址，以及数字签名的基础上。数字密钥实际上并不存储于网络中，而是由用户创建并以文件或者简单数据库的形式由用户自行保存，叫作钱包。用户钱包中的数字密钥是完全独立于比特币协议的，它可以由钱包软件创建、管理，而无须与区块链关联或者访问互联网。密钥的存在使比特币很多有意思的特性得以实现，包括去中心化的信用和控制、持有证明、加密安全模型等。

为了能够加入区块链中，每个比特币交易都需要提供一个有效签名，这个签名只能由有效的数字密钥产生；因而，任何持有密钥的人就拥有了这个账户的比特币控制权。密钥是成对出现的，包含一个私钥（需要保密）和一个公钥。我们可以把公钥想象成银行的账户代号，私钥则是控制这个账户的密码，或者支票上的用户签名。这些数字密钥极少被比特币用户看到。实际上，大多数时候，它们被存储在钱包文件中，并且由钱包软件进行管理。

在比特币的支付环节，接收者的公钥由其数字指纹替代，被称为比特币地址，它就像支票上的接收人名称（“收款人”）。大多数情况下，比特币地址是从公钥产生并与之关联的。但是比特币地址除了代表一个公钥，也可以代表其他“受益人”，比如稍后我们将在本章中见到的“脚本”。这种将资金接收方抽象为一个比特币地址的方式，使交易目标更为灵活，就像纸质支票：一笔支付指令既可用于向个人账户付款，也可以用于向公司账户付款；既可以账单支付，也可以现金支付。比特币地址是用户能看到密钥的唯一表现形式，因为它是需要与别人共享的部分。

在本章中，我们将介绍钱包软件，它管理着用户的密钥。我们将了解密钥是如何产生、存储和管理的。我们也将回顾各种用于公私钥、比特币地址、脚本地址的编码格式。最后，我们还将介绍一些密钥的特殊使用场景，如消息签名、所有权证明、创建“荣耀”地址、纸钱包等。

公钥密码学和加密货币

公钥密码学诞生于20世纪70年代，是计算机和信息安全的数学基础。

自公钥密码学发明以来，素数幂运算、椭圆曲线乘法等数学函数陆续被引入。这些函数都是不可逆的，也就是说，我们很容易从一个方向进行计算得出结果，但是想从结果倒推却是不可行的。由于这些数学函数的引入，数字密码和不可伪造数字签名的创建得以实现。比特币使用椭圆曲线乘法作为其公钥密码学的基础。

在比特币中，我们利用公钥密码学创建密钥对，以此来控制比特币的访问权。密钥对由私钥和公钥（由私钥派生而来）组成，公钥用于接收比特币，私钥用于对支付比特币的交易进行签名。

公钥与私钥在数学上的关系使私钥可用于生成消息签名。在不泄露私钥的情况下，公钥可以对该签名的有效性进行验证。

花费比特币时，当前比特币的持有人需要在交易的同时提供公钥和签名（签名每次不同，但都由相同的私钥创建）。通过附带的公钥和签名，比特币网络中的所有参与者都可以验证交易的有效性，并接受该笔交易，即确认该笔交易的资金在交易的发起时点确实是发起人所有的。

在大部分钱包的实现过程中，出于便利性，私钥和公钥是以密钥对的形式存储在一起的。但是由于公钥可以由私钥计算得来，因此只存

储私钥也是可行的。

私钥和公钥

一个比特币钱包通常包含一些密钥对的集合，每个密钥对包含一个私钥和一个公钥。私钥（k），是个数字，通常随机获取。从私钥出发，利用椭圆曲线乘法（一种单向加密函数），可以计算出一个公钥（K）。从公钥出发，利用单向加密哈希函数可以生成比特币地址（A）。在本节中，我们将首先研究如何生成私钥，然后研究生成公钥的椭圆曲线函数，最后从公钥生成一个比特币地址。私钥、公钥、比特币地址的关系见图4.1。

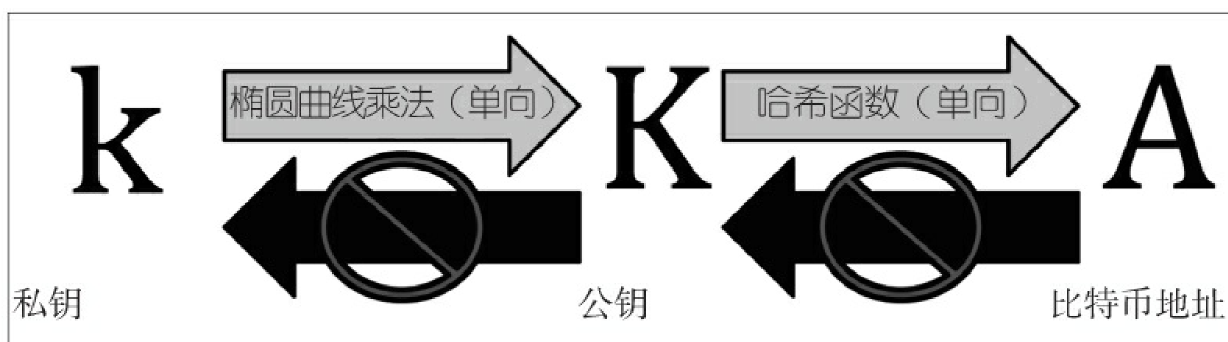


图4.1　私钥、公钥和比特币地址

私钥

一个私钥就是一串随机提取的数字，拥有和控制私钥是用户控制与比特币地址相关联的资金的根本。用户交易时想证明使用的资金是他自己的，必须使用其私钥对交易进行签名。在任何时候均必须保证私钥的私密性，将私钥透露给第三方，等同于把由它保护的比特币的控制权交给了第三方。私钥同样要进行备份、保护，防止意外丢失。如果私钥丢失，将是不可恢复的，受它保护的资金也就彻底丢失了。

比特币私钥只是一串数字。你可以利用抛硬币并用铅笔和纸张记录的方式来随机获取：抛256次硬币，你就获得了一个256位的二进制数字，这个数字可用作比特币钱包的私钥。生成私钥后，相应的公钥可以利用私钥计算得出。

从一个随机数生成私钥

要生成私钥，第一步也是最关键的一步是找到一个安全的熵源或者随机源。创建比特币密钥本质上就是“取得一个1到2^{256}之间的数字”。如果能保证随机数获取的方式是不可预测、不可重复的，则实际采用哪种方法无关紧要。比特币软件利用操作系统底层的随机数生成器来生成256比特的熵（随机数），通常操作系统的随机数是利用某种人工随机源进行初始化的，这也是为什么生成私钥的过程中会要求你随机晃动鼠标几秒钟。对于真正的偏执狂，投256次骰子，并且用铅笔和纸张记录下来更靠谱。

更准确地说，私钥是从1到n-1之间的任意数字，其中n是一个常量（n=1.158×10^{77}，这个数比2^{256}略小），在比特币中这个常量是作为椭圆曲线的幂来定义的（参见本章中的“椭圆曲线加密算法解释”）。为了生成这样一个密钥，我们随机取一个256位长度的数字，并验证其是否小于n-1。以程序的术语，这通常是从一个密码学安全的随机源中抽取一长段字符串，并通过SHA256哈希算法进行计算，这样就可以很方便地生成一个256比特长度的数字。如果上述步骤结果小于n-1，我们就得到了一个合适的私钥。否则，我们需要重复以上步骤，直到最终得到一个合适的私钥。

不要试图自己写代码生成密钥或者采用编程语言提供的“简单”的随机数生成器。采用密码学安全的伪随机数生成器（CSPRNG），并且从具有足够信息熵的来源中提取随机数种子。仔细研究你的随机数生成库的文档，以确定你选择的随机数生成器从密码学角度来讲是安全

的。正确选择的CSPRNG算法对于密钥的生成至关重要。

以下是以十六进制形式表示的随机生成的私钥（k）（256比特的二进制数字用十六进制表示共有64位，每位代表4比特）。

1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8

比特币私钥的可选范围是1~2^{256}，2^{256}是一个难以想象的大数，以十进制表示，它大概是10^{77}，而宇宙的可见部分，其组成也就大概10^{80}个原子。

为了使用比特币核心客户端（参见第3章）创建一个新的密钥，可以使用getnewaddress命令。出于安全考虑，命令输出只显示公钥，而不显示私钥。要让bitcoind进程暴露私钥，使用dumpprivkey命令。dumpprivkey命令以“Base58校验编码”（Base58 checksum-encoded）格式表示密钥，叫作钱包导入格式（**Wallet Import Format**，简称**WIF**），我们将在本章“密钥格式”中进一步详细介绍。以下例子展示了利用上述两个命令生成和显示私钥的步骤。

```
$ bitcoind getnewaddress
1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
$ bitcoind dumpprivkey 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

dumpprivkey命令打开钱包软件并且将getnewaddress生成的私钥解压。bitcoind是无法通过公钥知道私钥的，除非它们同时存储在钱包软件中。

dumpprivkey命令不是从公钥生成一个私钥，因为这根本不可能。这个命令只是简单地从“钱包”中取出由getnewaddress生成的私钥。

你也可以使用命令行工具sx（参见第3章“Libbitcoin和sx工具集”）来生成和展示私钥；相应的sx命令是newkey。

```
$ sx newkey
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

公钥

公钥是使用椭圆曲线乘法从私钥计算得来的，这是一个不可逆的过程:$K=k \times G$，其中，***k***是私钥，***G***是一个常数点（被称为生成点），***K***是计算结果，即公钥。其反向操作被称为“查找离散对数”——已知***K***，求***k***，其难度与尝试所有k的可能值的难度差不多，也就是说，与暴力求解基本等同。在演示如何从私钥生成公钥之前，我们先仔细看一下椭圆曲线加密算法。

椭圆曲线加密算法解释

椭圆曲线加密算法是一种非对称加密算法，或者叫公钥加密算法，它的基础是以椭圆曲线上点的加法运算或乘法运算表示的离散对数问题。

图4.2是一个椭圆曲线的例子，与比特币中使用的类似。

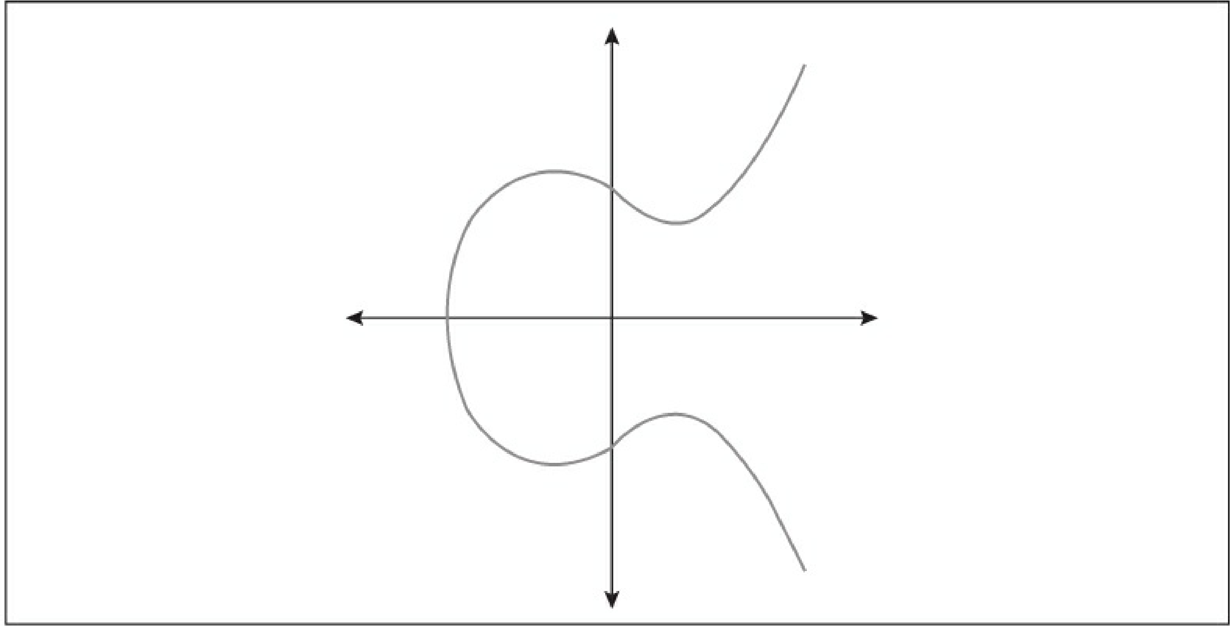


图4.2　一条椭圆曲线

比特币中使用的是一条特定的椭圆曲线和一系列数学常量，这些内容在美国国家标准技术研究所（National Institute of Standards and Technology，简称NIST）发布的secp256k1标准中进行了定义。secp256k1曲线是由以下函数定义的，是一条椭圆曲线：

$$y^2=(x^3+7) \text{ over } (\mathbb{F}_p)$$

或者

$$y^2 \bmod p=(x^3+7) \bmod p$$

mod p（对素数p取模）表明，这个曲线是在素数幂p的有限域上，也写成$\mathbb{F}_p$，其中$p=2^{256}-2^{32}-2^9-2^8-2^7-2^6-2^4-1$，是一个非常大的素数。

由于这条曲线是基于素数幂而不是基于实数有限域定义的，它的图像看起来像一堆散乱在两个象限上的点，很难画图表示。但是它在数学原理上与基于实数的椭圆曲线是一样的。作为一个例子，图4.3显示了

一个基于素数幂17（远小于实际值）的有限域上的椭圆曲线，可以看到一系列点散布在网格上。而secp256k1比特币椭圆曲线可以被想成一个在巨大网格上的更为复杂的散列点。

作为例子，我们选取secp256k1曲线上坐标为（x,y）的点P，使用Python来进行检验：

```
P =
(55066263022277343669578718895168534326250603453777594175500187360389116729240,
32670510020758816978083085130507043184471273380659243275938904335757337482424)
```

```
Python 3.4.0 (default, Mar 30 2014, 19:23:13)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> p =
115792089237316195423570985008687907853269984665640564039457584007908834671663
>>> x =
55066263022277343669578718895168534326250603453777594175500187360389116729240
>>> y =
32670510020758816978083085130507043184471273380659243275938904335757337482424
>>> (x ** 3 + 7 - y**2) % p
0
```

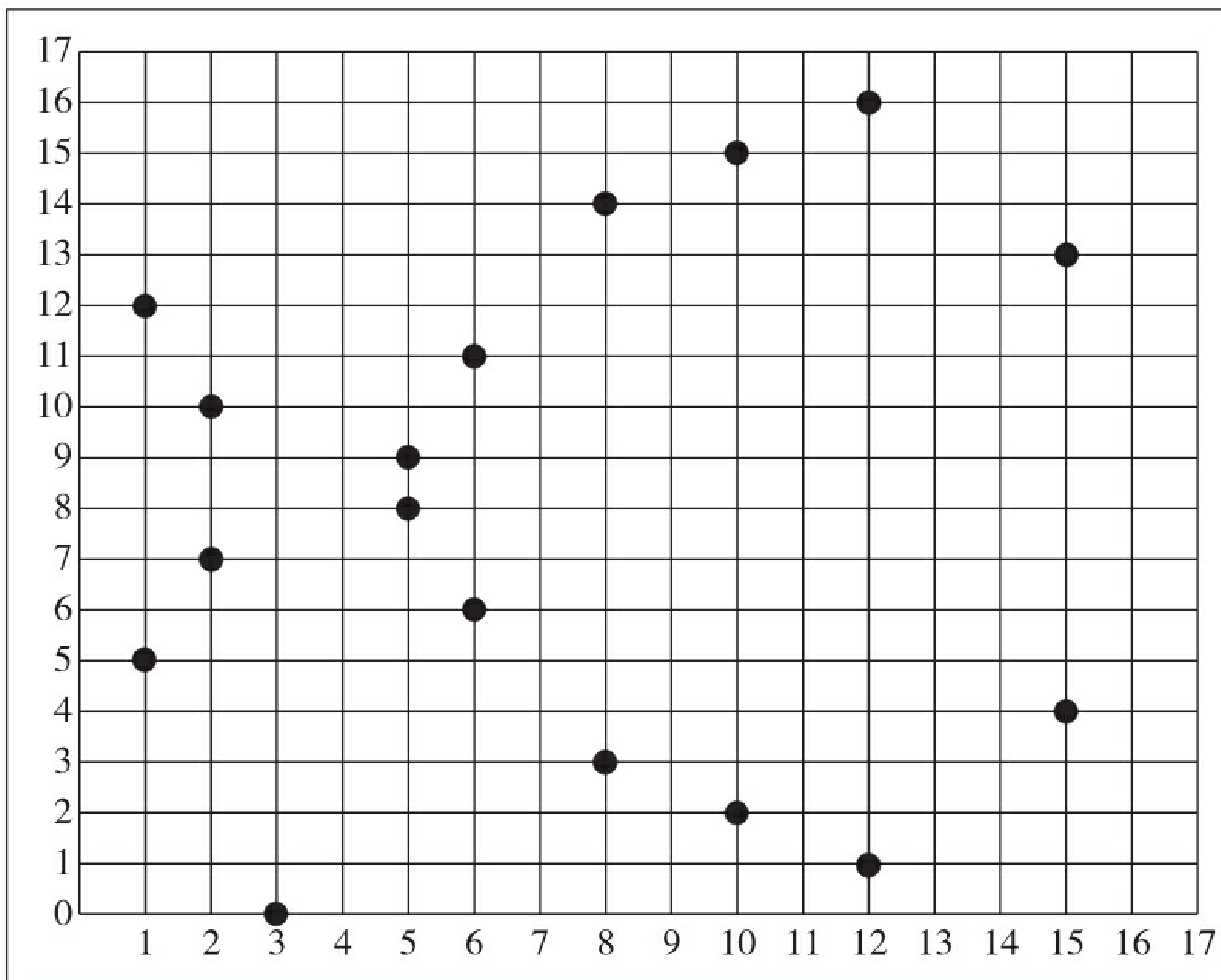


图4.3　椭圆曲线密码学：一个在F（p）上的椭圆曲线的图像，其中p=17

在椭圆曲线数学中，有个点叫作“无穷远点”，它与0在加法中扮演的角色大致相同。在计算机中，它有时也表示为x=y=0（虽然不满足椭圆曲线方程，但它是一个简单可校验的独立案例）。

还有个“+”号运算符，叫作“加法”，它有点类似于小时候学过的传统实数加法。给定在椭圆曲线上的两个点P_1和P_2，存在第三个点$P_3=P_1+P_2$，也在椭圆曲线上。

在几何学上，这个点P_3是通过在P_1和P_2间绘制一条直线来计算的。这条直线将与椭圆曲线相交于一点P'_3=（x,y），通过x轴映射，得到

P_3=（x,-y）。

有很多特殊案例解释了“无穷远点”存在的必要性。

如果P_1和P_2是同一点，那P_1和P_2的连接线必然与曲线在P_1点相切，曲线有且仅有一个新的点与直线相交。可以使用微积分方法来计算切线的斜率。虽然我们感兴趣的点被限制为曲线上两个坐标均为整数的点，这些方法仍然能以奇怪的方式满足要求！

在某些情况下（比如：P_1和P_2的x值相同，但y值不同），切线将是垂直的，则P_3=“无穷远点”。

如果P_1是无穷远点，那么P_1+P_2=P_2；相应地，如果P_2是无穷远点，那么P_1+P_2=P_1。这显示了其与0一样的性质。

事实证明，这里的“+”号符合联合率，也就是说（A+B）+C=A+（B+C）。这意味着，我们可以不带括号将其写成A+B+C，而不会有任何歧义。

至此，我们已经定义了加法，我们也可以按照标准的方式通过扩展加法来定义乘法。对于椭圆曲线上的点P，如果k是个整数，那么kP=P+P+P+…+P（k次）。需要注意的是，在这种情况下，k有时也被称为“指数”。

生成一个公钥

从一个密钥（形式上是一个随机生成的数字k）开始，我们将它与曲线上预定义的点相乘，可以得到曲线上的另一个点，这就是相应的公钥K，而这个预定义的点叫作生成点**G**。生成点是作为secp256k1标准的一部分定义的，对于比特币而言，其所有密钥均使用相同G点。

K=k×G

这里，k是密钥，G是生成点，K是生成的公钥，也是椭圆曲线上的一个点。由于生成点对所有比特币用户来说都是一样的，一个密钥k与G相乘后总能得到相同的公钥K。k与K之间的关系是固定的，但是只能从k到K进行单向计算。这也是比特币地址（从K衍生而来）可以与任何人共享，却不会暴露用户私钥（k）的原因。

私钥可以转换为公钥，但是公钥不能转换回私钥，因为从数学的角度，这种计算是单向的。

为实现椭圆曲线乘法，我们使用之前生成的私钥k，与生成点G相乘，以获得公钥K：

```
K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD * G
```

公钥K定义成一个点K=（x,y）：

K=（x,y）

where，

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

为了形象化演示一个点与整数的相乘，我们使用一个简单的基于实数的椭圆曲线——记住，实数与整数在数学上都是一样的。我们的目标是找到生成点G的倍数kG。也就是G相加k次。在椭圆曲线中，一个点与其自身相加等同于在这个点上画一条切线，找到切斜与曲线相交的点，相交点相对x轴对称的点就是我们要找的点。

图4.4显示了如何利用几何学在曲线上获得G，2G，4G。

大多数比特币实现都利用OpenSSL加密库（http://bit.ly/1ql7bn8）来完成椭圆曲线算法的计算。比如，为了获得公钥，就会用

到EC_POINT_mul（）函数。

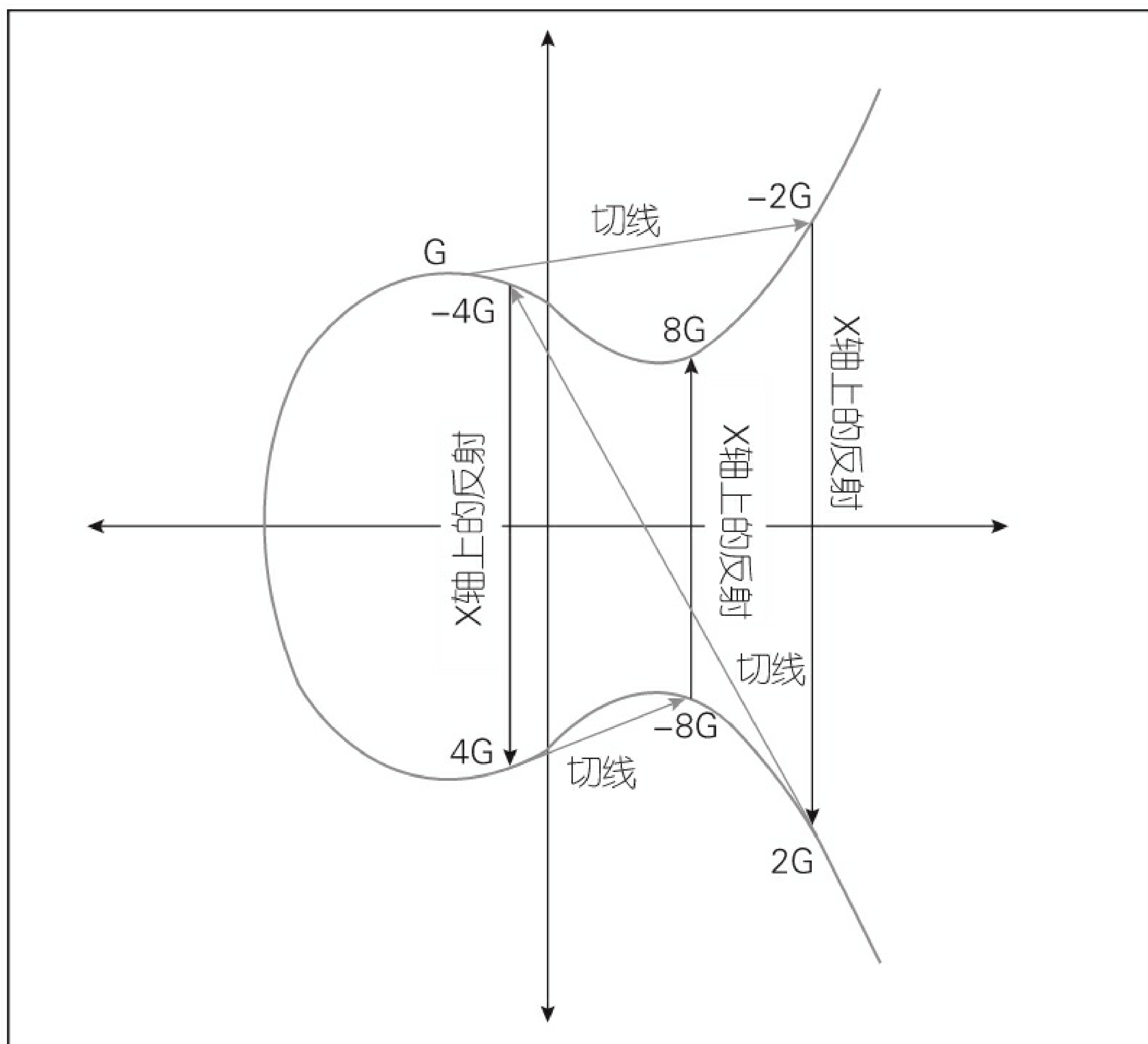


图4.4　椭圆曲线加密算法：演示G在椭圆曲线上被整数k相乘

比特币地址

比特币地址是一串由数字和字母构成的字符串，可以分享给任何想给你转钱的人。地址是从公钥转换而来的，包含数字和字母，其第一个字符是1（数字）。下面是一个比特币地址的例子。

1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

比特币地址在交易中通常以资金“接收者”的形式出现。如果我们将比特币的交易与纸质支票做个类比，那么比特币地址就相当于支票上的受益人，也就是我们写在支票“收款人”后的内容。在纸质支票上，受益人可以是一个银行账户持有者的姓名，也可以是公司、机构，甚至现金。由于支票不需要指定账号，只是使用一个抽象的名字作为资金接收人，因此作为支付工具，支票非常灵活。比特币的交易采用了类似的抽象方法，即比特币地址，使其同样极具灵活性。比特币地址可以指代一个密钥对的拥有者，也可以指代其他一些东西，比如支付脚本。我们将在本章“支付到脚本哈希（P2SH）与多重签名地址”中讲到。现在，我们通过一个简单例子，了解代表公钥的比特币地址是如何由公钥产生的。

比特币地址是通过一种单向的加密哈希算法从公钥推导出来的。“哈希算法”是一种单向函数，它可以对任意长度的输入进行计算，生成输入信息的指纹或者“哈希”。加密哈希函数在比特币中应用广泛，包括比特币地址、脚本地址，以及挖矿中的工作量证明算法等。用于从公钥创建比特币地址的算法是安全哈希算法（SHA）和RACE完整性原语求值信息摘要算法（RIPEMD），应用的是其中两种特定算法，SHA256和RIPEMD160。

我们从公钥K开始，计算它的SHA256哈希值，然后从其结果中计

算RIPEMD160的哈希值，这样我们就创建了一个160比特（20字节）长度的数字：

A=RIPEMD160（SHA256（K））

其中，K是公钥；A是计算结果，即比特币地址。

比特币地址与公钥不一样，比特币地址是利用单向哈希函数从公钥计算得来的。

比特币地址基本上是以Base58Check编码形式（参见本章中“Base58和Base58Check编码”）展现给用户的，它使用58个字符（一个Base58数字系统）和一个校验码，能够达到方便阅读、避免歧义、防止地址转录及输入时犯错的效果。Base58Check在比特币中还有很多其他应用，比如比特币地址、用户私钥、加密的密钥、脚本哈希等。在下一节，我们将看到Base58Check的编码和解码机制，以及它们的结果展示。图4.5描述了从公钥到比特币地址的转换过程。

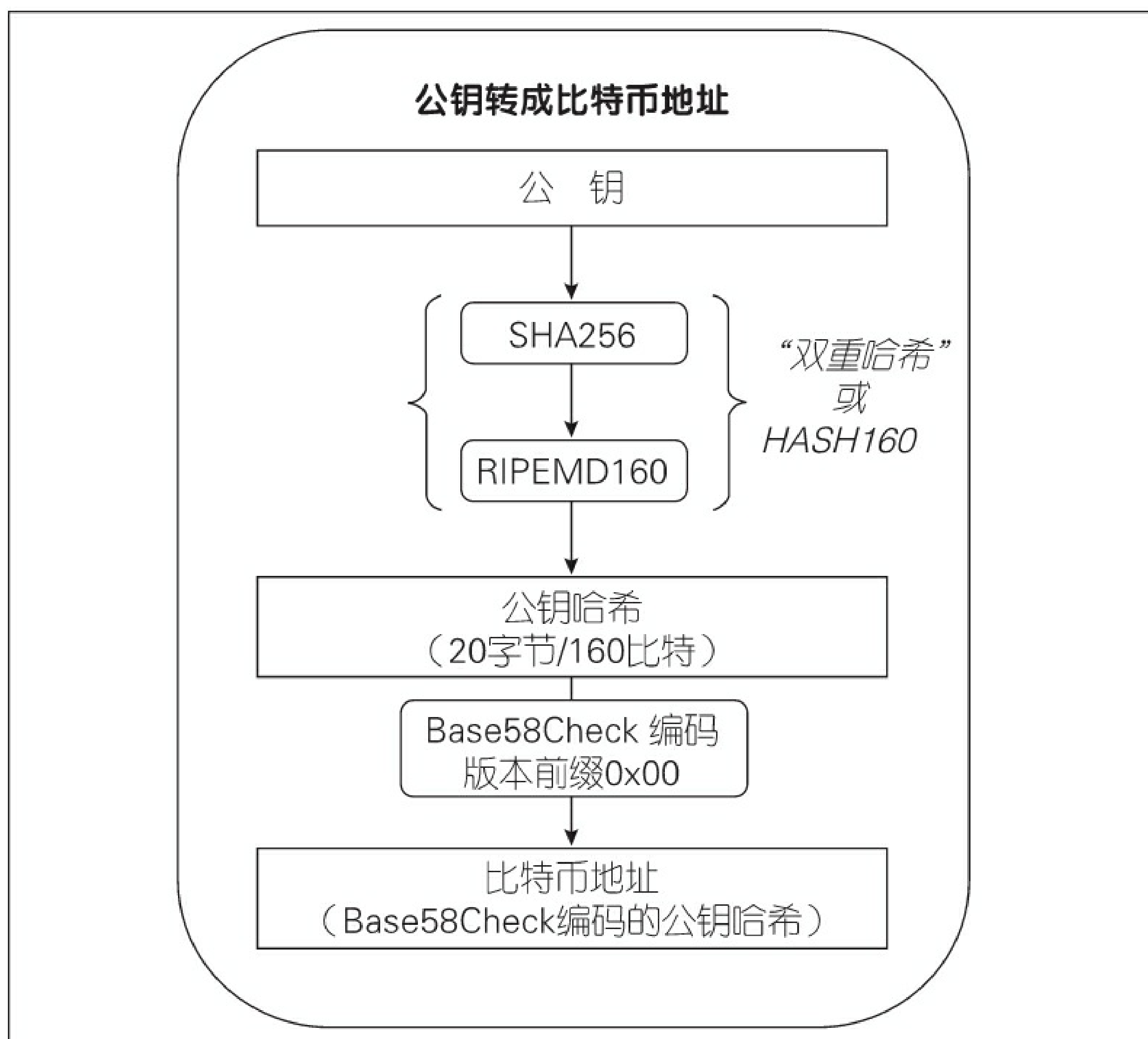


图4.5　从公钥到比特币地址：将公钥转换为比特币地址

Base58和Base58Check编码

为了以更少的符号、更紧凑的方式表示一个很大的数，很多计算机系统采用超过十的进制（底数），并混合使用字母和数字的表示法。举例来说，传统十进制系统使用10个从0到9的数字字符，十六进制系统使用16个字符，包括10个数字字符和A到F的6个字母。数字采用十六进制表示的话，就会比用十进制表示来得短。Base-64编码采用26个小写字

母、26个大写字母、10个数字字符，以及两个额外的字符，比如“+”“/”，对数据进行编码，以实现在基于文本的媒介——比如电子邮件上传输二进制数据。Base-64主要用于电子邮件中二进制附件的编码。Base58也是一个基于文本的二进制编码格式，用于比特币及很多其他密码货币系统中。它在紧凑性、可读性、错误检测与预防方面提供了一种平衡。Base58是Base64的一个子集，使用大小写字母和数字，但是省略了一些容易混淆的字符。具体来说，Base58是Base64编码中去掉0（数字零）、O（字母o的大写）、l（小写的L）、I（大写的i）、字符“+”和字符“/”。或者更简单地说，它是去掉四个字符（0,O,l,I）之后的大小写字母与数字的集合。

例**4-1**　比特币的**Base58**字符表

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
```

为了增加额外安全性，防止打字和转录出现错误，常用于比特币的Base58Check编码格式在Base58的基础上增加了内置的错误校验码。校验码为4个字节长，添加到需要编码的数据后面。校验码从待编码数据的哈希值得出，从而可以检测和避免转录和输入错误。取得一个Base58Check编码的数据后，解码软件可以计算原始数据的校验码，并与数据中的校验码进行比对。如果两者不一致，就说明原始数据有误，Base58Check数据无效。在比特币的实践中，这样可以避免钱包应用中将输错的比特币地址当作一个有效目标，防止资金丢失。

为了将数据（一个数字）转换为Base58Check编码格式，我们首先要添加一个前缀到数据前，称之为“版本字节”，这可以让我们更容易判断数据的类型。例如，比特币地址的前缀是0（十六进制表示为0x00），而私钥的前缀为128（十六进制表示为0x80）。常用版本前缀请参看表4.1。

接下来，我们计算“双重SHA”校验码，即对前面得到的数据（前缀+数据）进行两次SHA256哈希计算：

```
checksum = SHA256(SHA256(prefix+data))
```

从结果的32字节哈希值（哈希的哈希），我们只取前面4个字节。这4个字节作为错误检查码，或者校验码附加到数据的最后。

这样，就形成了由3部分（前缀、原始数据、校验码）组成的数据。现在可以利用前面介绍的Base58字符表，对结果数据进行编码。图4.6描述了Base58Check的编码过程。

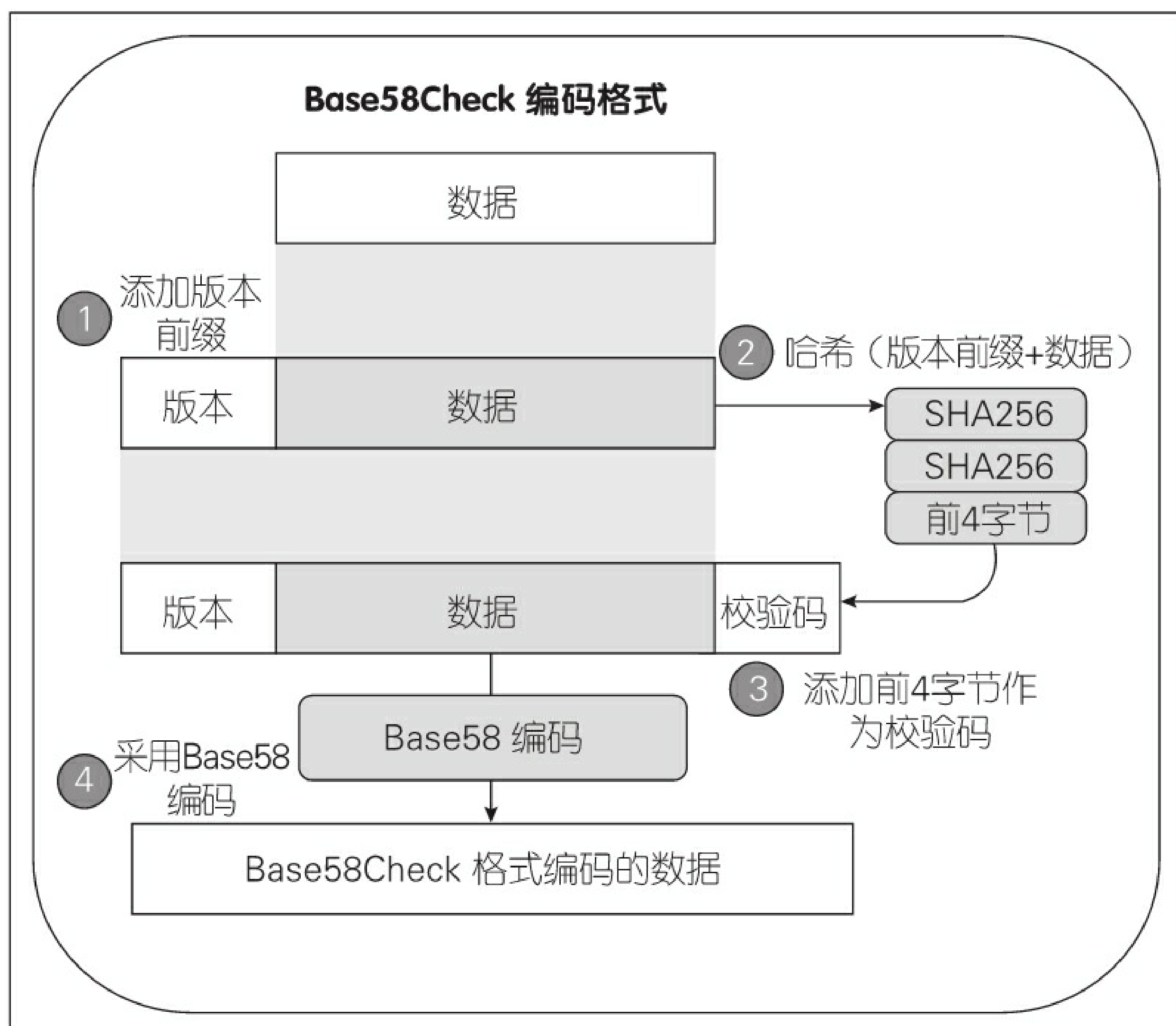


图4.6　Base58Check编码：一种基于Base58的、版本化的、可校验的比特币数据无歧义编码格式

在比特币中，大多数需要向用户展示的数据均以Base58Check格式

进行编码，这使数据紧凑、易读、便于检查错误。Base58Check编码的版本前缀用于创建易于辨别的格式，这意味着，当以Base58编码时，使用Base58Check编码后的数据头部包含了特定的字符。这个字符使用户很容易地判断出数据类型，以及如何去使用它。比如，Base58Check编码的比特币地址以1开头，而Base58Check编码的WIF格式私钥以5开头。一些版本前缀的例子，及其编码后的Base58字符见表4.1。

表4.1　Base58Check版本前缀及编码后结果示例

类型	版本前缀（十六进制）	Base58 结果前缀
比特币地址	0x00	1
脚本支付地址	0x05	3
比特币测试网地址	0x6F	m 或 n
私钥钱包导入格式	0x80	5、K 或 L
BIP38 加密私钥	0x0142	6P
BIP32 扩展公钥	0x0488B21E	xpub

我们来看一下完整的比特币地址生成过程，从私钥到公钥（椭圆曲线上的一个点），到双重哈希的地址，到最后Base58Check编码。例4-2中的C++代码逐步展示了从私钥一直到Base58Check编码的比特币地址的完整过程。代码使用了libbitcoin库中的一些函数（第3章“替代客户端、库、工具集”中介绍过）。

例**4-2**　从私钥创建**Base58Check**编码的比特币地址

```
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Private secret key.
    bc::ec_secret secret = bc::decode_hash(
        "038109007313a5807b2eccc082c8c3fbb988a973cacf1a7df9ce725c31b14776");
    // Get public key.
    bc::ec_point public_key = bc::secret_to_public_key(secret);
    std::cout << "Public key: " << bc::encode_hex(public_key) << std::endl;

    // Create Bitcoin address.
    // Normally you can use:
    //    bc::payment_address payaddr;
    //    bc::set_public_key(payaddr, public_key);
    //    const std::string address = payaddr.encoded();

    // Compute hash of public key for P2PKH address.
    const bc::short_hash hash = bc::bitcoin_short_hash(public_key);

    bc::data_chunk unencoded_address;
    // Reserve 25 bytes
    //    [ version:1  ]
    //    [ hash:20    ]
    //    [ checksum:4 ]
    unencoded_address.reserve(25);
    // Version byte, 0 is normal BTC address (P2PKH).
    unencoded_address.push_back(0);
    // Hash data
    bc::extend_data(unencoded_address, hash);
    // Checksum is computed by hashing data, and adding 4 bytes from hash.
    bc::append_checksum(unencoded_address);
    // Finally we must encode the result in Bitcoin's base58 encoding
    assert(unencoded_address.size() == 25);
    const std::string address = bc::encode_base58(unencoded_address);

    std::cout << "Address: " << address << std::endl;
    return 0;
}
```

代码使用预定义好的私钥，使每次运行都可以得到相同的比特币地址，就像例4-3所展示的一样。

例**4-3**　编译运行这段代码

```
# Compile the addr.cpp code
$ g++ -o addr addr.cpp $(pkg-config --cflags --libs libbitcoin)
# Run the addr executable
$ ./addr
Public key: 0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa
Address: 1PRTTaJesdNovgne6Ehcdu1fpEdX7913CK
```

密钥格式

不管私钥还是公钥，都可以表示为一系列不同的格式。尽管它们看起来并不一样，这些表现形式均是对同样数字的编码。这些格式的主要作用在于方便用户阅读及密钥转录，避免出现错误。

私钥格式

私钥可以表示为几种不同的格式，所有格式均代表与之对应的相同的256比特长的数字。表4.2显示了三种用于表示私钥的常用格式。

表4.2　私钥表示形式（编码格式）

类型	前缀	说明
Hex	无	64 位十六进制数字
WIF	5	Base58Check 编码：带 128 前缀和 32 比特校验码的 Base58 编码
WIF-compressed	K 或 L	同上，但编码前加一个 0x01 后缀

表4.3通过这三种格式展示了同一个私钥。

表4.3　例子：相同密钥、不同格式

格式	私钥
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

以上这些是用不同格式对相同密钥编码后的结果。虽然看起来不同，但是一种编码格式可以很容易地转换为另一种格式。

从**Base58Check**到十六进制解码

使用sx工具包（参看第3章“libbitcoin和sx工具集”），我们可以很容易地写出脚本或者命令行“管道”来操控比特币密钥、地址和交易。你可以使用sx工具包通过命令行来解码Base58Check格式。

我们使用base58check-decode命令解码未压缩密钥：

```
$ sx base58check-decode 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd 128
```

结果是一个十六进制格式的密钥，跟着一个钱包导入格式（WIF）的版本前缀128。

从十六进制到**Base58Check**编码

将密钥编码为Base58Check格式（与前述命令刚好相反），我们提供十六进制的私钥，跟着一个钱包导入格式（WIF）的版本前缀128。

```
$ sx base58check-encode
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd 128
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

从十六进制（压缩格式密钥）到**Base58Check**的编码

为了将“压缩”格式的私钥（参见本章中“压缩私钥”）编码成

Base58Check格式，我们在十六进制密钥的最后加上后缀01，然后进行编码：

```
$ sx base58check-encode
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01 128
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

生成的WIF-压缩格式结果，以字母“K”开头。这表示私钥带有“01”后缀，并且只可用于生成压缩格式的公钥（参见本章中“压缩公钥”）。

公钥格式

公钥也同样可以采用不同的格式表示，其中最重要的是压缩和非压缩公钥格式。

就像我们前面看到的，公钥是一个在椭圆曲线上的点，包含一对坐标（x,y）。它通常的表现形式为：04前缀开头，紧跟两个256位长度的数字，一个代表*x*坐标，另一个代表*y*坐标。04前缀用于区别非压缩公钥和压缩公钥，压缩公钥是以02或03开头的。

以下是之前我们通过私钥生成的公钥，以x,y坐标表示。

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

这里是以520比特（130个十六进制数字）的数字显示的相同密钥，04开头，紧跟着x坐标和y坐标，即04 x y。

```
K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A<?pdf-cr?
>07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

压缩公钥

比特币采用压缩公钥的目的是降低交易文件的大小，使存储比特币区块链数据库的节点尽可能地节省磁盘空间。大多数交易均包含公钥，用以验证所有者的身份并花费比特币。每个公钥的长度为520比特（前缀\\+*x*\\+*y*），而每个区块由几百个交易组成，每天都有成千上万个交易

加入区块，这给区块链的存储带来了一定负担。

我们在本章“公钥”中看到，公钥其实是椭圆曲线上的一个点。因为曲线代表了一个数学方程，曲线上的一个点就代表了方程的一个解，那么，如果我们知道*x*坐标，*y*坐标就能通过求解方程$y^2 \bmod p=(x^3+7) \bmod p$得到。因此我们也可以只存储公钥中的*x*坐标，而把*y*坐标省略掉，这样就将所需空间减少了256比特，几乎少了一半。长期来看，交易过程中节省的空间将是相当可观的。

未压缩公钥带04前缀，而压缩公钥始于02或者03。我们看看为什么会有两个不同的前缀：方程的左边是y^2，这意味着*y*的解是个平方根，可以是正值也可以是负值。或者说，结果中*y*坐标可能位于*x*轴之上，也可能位于*x*轴之下。就如我们在图4.2看到的曲线图像，它是相对x轴对称的。所以，当我们省略*y*坐标时，我们必须保留*y*的符号（正还是负），换言之，我们必须记住这个点是在*x*轴之上还是*x*轴之下，上下两个点代表了两个不同的公钥。当我们在素数幂*p*的有限域内以二进制算术计算椭圆曲线的时候，*y*坐标可能是偶数也可能是奇数，分别对应*y*坐标的正负符号。这样，为了区分两个可能的*y*值，我们在压缩公钥中用前缀02代表偶数，03代表奇数，从而保证软件可以从*x*坐标准确推断出*y*坐标，并将公钥解压成完整坐标的点。公钥压缩过程参见图4.7。

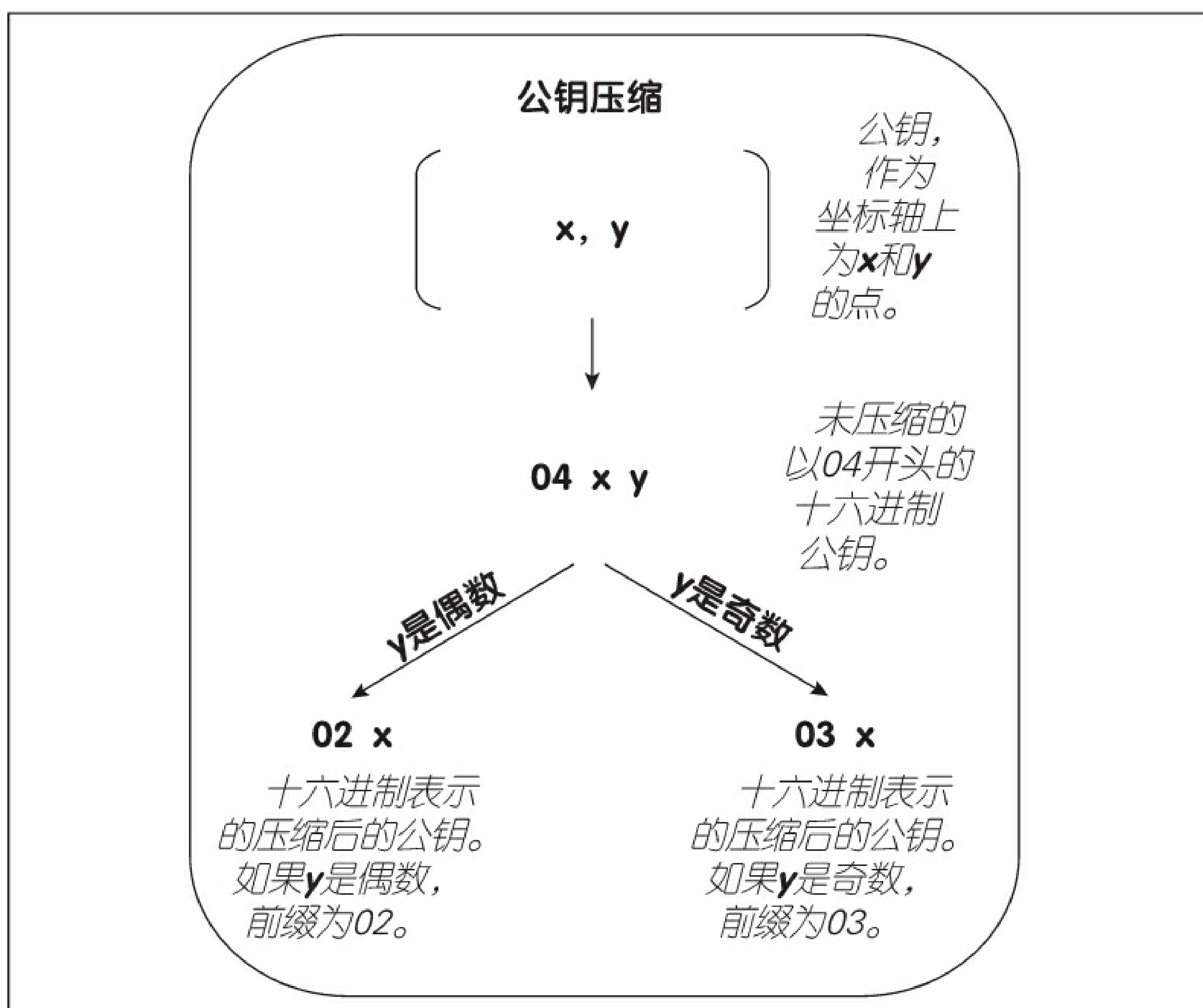


图4.7　公钥压缩

这是前面生成的、用压缩格式存储的264比特（66个十六进制字符）公钥，以03为前缀，代表*y*坐标是奇数：

```
K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
```

虽然它看起来与非压缩公钥并不一样，但对应的都是相同的私钥，也就是从相同的私钥生成的。更重要的是，如果我们将一个压缩公钥使用双重哈希函数（RIPEMD160（SHA256（K）））转换为一个比特币地址时，将生成一个不同的地址。这就很容易引起混淆，因为这意味着一个私钥可以生成两种不同表达方式的公钥（压缩、非压缩），进一步还将生成两个不同的比特币地址。但是对应这两个不同比特币地址的私

钥却是相同的。

压缩公钥已逐渐成为比特币客户端的默认配置，这对降低交易文件大小，进而降低区块链大小有一定的积极作用。但是，并不是所有的客户端都支持压缩公钥格式。支持压缩公钥的新版客户端必须能兼容不支持压缩公钥的老版客户端发来的交易。这对于从其他钱包应用中导入私钥尤为重要，因为新钱包需要扫描区块链以查找与这些私钥相关的所有交易。比特币钱包到底该扫描哪种类型的地址呢？非压缩公钥生成的地址还是压缩公钥生成的地址？这两种地址都是有效的比特币地址，它们都可以被私钥签名，但它们的确是两个不同的地址！

为了解决这个麻烦，当私钥从钱包中导出时，用于表示私钥的钱包导入格式（WIF）在新比特币钱包中采用了不同的实现方式，它可以指示私钥已经被用于创建压缩公钥，并且也生成了压缩比特币地址。由此导入钱包可以辨别私钥是从旧的钱包来的，还是从新的钱包来的，从而根据非压缩还是压缩的比特币地址从区块链上搜索交易。我们将在下一节来看这个过程的细节。

压缩私钥

有意思的是，名词“压缩私钥”有点误导的意味，实际上私钥以WIF-压缩格式导出时比那些“未压缩”的私钥还长了1个字节。因为它加了一个01的后缀，这个后缀表明它是从一个新钱包来的，只能用于生成压缩公钥。私钥既没有被压缩，也不可能被压缩。“压缩私钥”的真正含义是“只能用于生成压缩公钥的私钥”；同样，“非压缩私钥”就是“只能生成非压缩公钥的私钥”。为避免带来更多混淆，你最好将导出格式称为“WIF-压缩格式”或者“WIF”，而不是将私钥称为“压缩的”或者“非压缩的”。

记住，这些格式不是可交换的。在实现了压缩公钥的新钱包中，私钥只能被转换为WIF-压缩格式公钥（带K或L前缀）。如果钱包是一个

较老版本的实现，还不能支持压缩公钥，那么，私钥只能导出为WIF格式（带5前缀）。这样做的目的是通知导入这些私钥的钱包，究竟是以压缩公钥以及它们相应的比特币地址进行搜索，还是以非压缩公钥及其地址进行搜索。

如果比特币钱包支持压缩公钥，它将在所有交易中使用压缩公钥。钱包中的私钥用于在曲线中生成公钥点，这个公钥点将进行压缩。压缩后的公钥用于生成比特币地址，并应用在交易当中。从一个支持压缩公钥的钱包导出私钥时，钱包导入格式将被修改，添加一个01后缀到私钥中。经过Base58Check编码后的私钥被称为“压缩WIF”，始于字母K或L，而不是像使用WIF编码（未压缩）的老钱包应用一样始于数字“5”。同一个密钥，以WIF和WIF-压缩格式两种不同格式编码如表4.4所示。

表4.4　同一个密钥，以WIF和WIF-压缩格式两种不同格式编码

格式	私钥
十六进制	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
十六进制－压缩	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD_01_
WIF－压缩	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZ-gawvrtJ

“压缩私钥”是一个误会！它们并没有被压缩；相反地，WIF-压缩格式只是表明它们只能用于生成压缩公钥以及相应的比特币地址。有意思的是，一个“WIF-压缩格式”编码的私钥比“不压缩”的格式还长一个字节，因为它被加上了一个“01”的后缀，用于与“非加密”进行区分。

在**Python**中实现密钥和地址

最全的Python比特币库是维塔里克·布特林（Vitalik Buterin）开发的pybitcointools（**http://github.com/vbuterin/pybitcointools**）。在例4-4中，我们利用这个库（import时使用bitcoin）来生成密钥和地址，并以不同格式进行展示。

例**4-4**　利用**pybitcointools**库生成并格式化密钥和地址

```
import bitcoin

# Generate a random private key
valid_private_key = False
while not valid_private_key:
    private_key = bitcoin.random_key()
    decoded_private_key = bitcoin.decode_privkey(private_key, 'hex')
    valid_private_key =  0 < decoded_private_key < bitcoin.N

print "Private Key (hex) is: ", private_key
print "Private Key (decimal) is: ", decoded_private_key

# Convert private key to WIF format
wif_encoded_private_key = bitcoin.encode_privkey(decoded_private_key, 'wif')
```

```
print "Private Key (WIF) is: ", wif_encoded_private_key

# Add suffix "01" to indicate a compressed private key
compressed_private_key = private_key + '01'
print "Private Key Compressed (hex) is: ", compressed_private_key

# Generate a WIF format from the compressed private key (WIF-compressed)
wif_compressed_private_key = bitcoin.encode_privkey(
    bitcoin.decode_privkey(compressed_private_key, 'hex'), 'wif')
print "Private Key (WIF-Compressed) is: ", wif_compressed_private_key

# Multiply the EC generator point G with the private key to get a public key point
public_key = bitcoin.base10_multiply(bitcoin.G, decoded_private_key)
print "Public Key (x,y) coordinates is:", public_key

# Encode as hex, prefix 04
hex_encoded_public_key = bitcoin.encode_pubkey(public_key,'hex')
print "Public Key (hex) is:", hex_encoded_public_key

# Compress public key, adjust prefix depending on whether y is even or odd
(public_key_x, public_key_y) = public_key
if (public_key_y % 2) == 0:
    compressed_prefix = '02'
else:
    compressed_prefix = '03'
hex_compressed_public_key = compressed_prefix + bitcoin.encode(public_key_x, 16)
print "Compressed Public Key (hex) is:", hex_compressed_public_key

# Generate bitcoin address from public key
print "Bitcoin Address (b58check) is:", bitcoin.pubkey_to_address(public_key)

# Generate compressed bitcoin address from compressed public key
print "Compressed Bitcoin Address (b58check) is:", \
    bitcoin.pubkey_to_address(hex_compressed_public_key)
```

例4-5显示这段代码的运行结果。

例**4-5**　运行**key-to-address-ecc-example.py**的结果

```
$ python key-to-address-ecc-example.py
Private Key (hex) is:
 3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa6
Private Key (decimal) is:
 26563230048437957592232553826663696440606756685920117476832299673293013768870
Private Key (WIF) is:
 5JG9hT3beGTJuUAmCQEmNaxAuMacCTfXuw1R3FCXig23RQHMr4K
Private Key Compressed (hex) is:
 3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa601
Private Key (WIF-Compressed) is:
 KyBsPXxTuVD82av65KZkrGrWi5qLMah5SdNq6uftawDbgKa2wv6S
Public Key (x,y) coordinates is:
 (41637322786646325214887832269588396900663353932545912953362782457239403430124L,
 16388935128781238405526710466724741593761085120864331449066658622400339362166L)
Public Key (hex) is:
 045c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec↵
243bcefdd4347074d44bd7356d6a53c495737dd96295e2a9374bf5f02ebfc176
Compressed Public Key (hex) is:
 025c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec
Bitcoin Address (b58check) is:
 1thMirt546nngXqyPEz532S8fLwbozud8
Compressed Bitcoin Address (b58check) is:
 14cxpo3MBCYYWCgF74SWTdcmxipnGUsPw3
```

例4-6是另一个例子，使用Python的ECDSA库来计算椭圆曲线，没有使用任何特定的比特币库。

例**4-6**　演示比特币密钥中使用的椭圆曲线数学的脚本

```
import ecdsa
import random
from ecdsa.util import string_to_number, number_to_string

# secp256k1, http://www.oid-info.com/get/1.3.132.0.10
_p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2FL
_r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141L
_b = 0x0000000000000000000000000000000000000000000000000000000000000007L
_a = 0x0000000000000000000000000000000000000000000000000000000000000000L
_Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798L
_Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8L
curve_secp256k1 = ecdsa.ellipticcurve.CurveFp(_p, _a, _b)
generator_secp256k1 = ecdsa.ellipticcurve.Point(curve_secp256k1, _Gx, _Gy, _r)
oid_secp256k1 = (1, 3, 132, 0, 10)
SECP256k1 = ecdsa.curves.Curve("SECP256k1", curve_secp256k1, generator_secp256k1,
oid_secp256k1)
ec_order = _r

curve = curve_secp256k1
generator = generator_secp256k1

def random_secret():
    random_char = lambda: chr(random.randint(0, 255))
    convert_to_int = lambda array: int("".join(array).encode("hex"), 16)
    byte_array = [random_char() for i in range(32)]
    return convert_to_int(byte_array)

def get_point_pubkey(point):
    if point.y() & 1:
        key = '03' + '%064x' % point.x()
    else:
        key = '02' + '%064x' % point.x()
    return key.decode('hex')

def get_point_pubkey_uncompressed(point):
    key = '04' + \
          '%064x' % point.x() + \
```

```
        '%064x' % point.y()
    return key.decode('hex')

# Generate a new private key.
secret = random_secret()
print "Secret: ", secret

# Get the public key point.
point = secret * generator
print "EC point:", point

print "BTC public key:", get_point_pubkey(point).encode("hex")

# Given the point (x, y) we can create the object using:
point1 = ecdsa.ellipticcurve.Point(curve, point.x(), point.y(), ec_order)
assert point1 == point
```

例4-7显示了运行这个脚本后所产生的输出。

例**4-7**　安装**Python ECDSA**库并运行**ec_math.py**脚本

```
$ # Install Python PIP package manager
$ sudo apt-get install python-pip
$ # Install the Python ECDSA library
$ sudo pip install ecdsa
$ # Run the script
$ python ec-math.py
Secret:
38090835015954358862481132628887443905906204995912378278060168703580660294000
EC point:
(70048853531867179489857750497606966272382583471322935454624595540007269312627,
105262206478686743191060800263479589329920209527285803935736021686045542353380)
BTC public key: 029ade3effb0a67d5c8609850d797366af428f4a0d5194cb221d807770a1522873
```

钱包

钱包是保存私钥的容器，通常以结构化文件或者简单数据库的方式实现。另一个生成私钥的方式是确定性密钥生成。使用确定性密钥生成的情况下，可以通过单向哈希函数，从上一个私钥中生成一个新的私钥，按顺序连接，形成一个链条。如果需要重建这个链条，你只需生成第一个私钥（称之为种子或者主密钥），即可生成整个序列。在本节中，我们将检查不同的密钥生成方式以及相应的钱包结构。

比特币钱包包含的是密钥，而不是比特币。每个用户拥有一个包含很多密钥的钱包。钱包实际上是一个密钥链，包含一对对的公/私钥（参看本章中“私钥和公钥”）。用户使用密钥对交易进行签名，以证明其拥有交易输出（比特币）。比特币以交易输出的方式存储于区块链上（通常记为vout或者txout）。

非确定性（随机）钱包

在早期的比特币客户端中，钱包是随机生成的私钥的简单集合。这种类型的钱包称为**Type-0**非确定性钱包。举例来说，比特币核心客户端在第一次启动时预生成100个随机私钥，后面则根据需要继续产生，每个私钥只使用一次。这种类型的钱包被戏称为“只是一堆密钥（Just a Bunch Of Keys）”，或者简写为“JBOK”，这种钱包已逐渐被确定性钱包替代，因为它们很难管理、备份和导入。随机密钥的缺点在于如果你生成了太多的密钥，就必须经常对所有这些密钥进行备份。如果没有备份，一旦钱包无法访问，这些密钥控制的资金将彻底丢失。这也与每个地址仅在交易中使用一次、避免地址重用的原则直接冲突。地址重用使

他人能够通过将多个交易和地址互相关联，从而获取用户的隐私信息。使用Type-0非确定型钱包是一个无奈的选择，特别是你为了避免地址重用而不得不使用大量密钥时，这使得频繁备份成为必要。虽然比特币核心包含一个Type-0钱包，但比特币核心的开发者却不建议使用这个钱包。图4.8展示了一个非确定型钱包，包含一些随机密钥的松散组合。

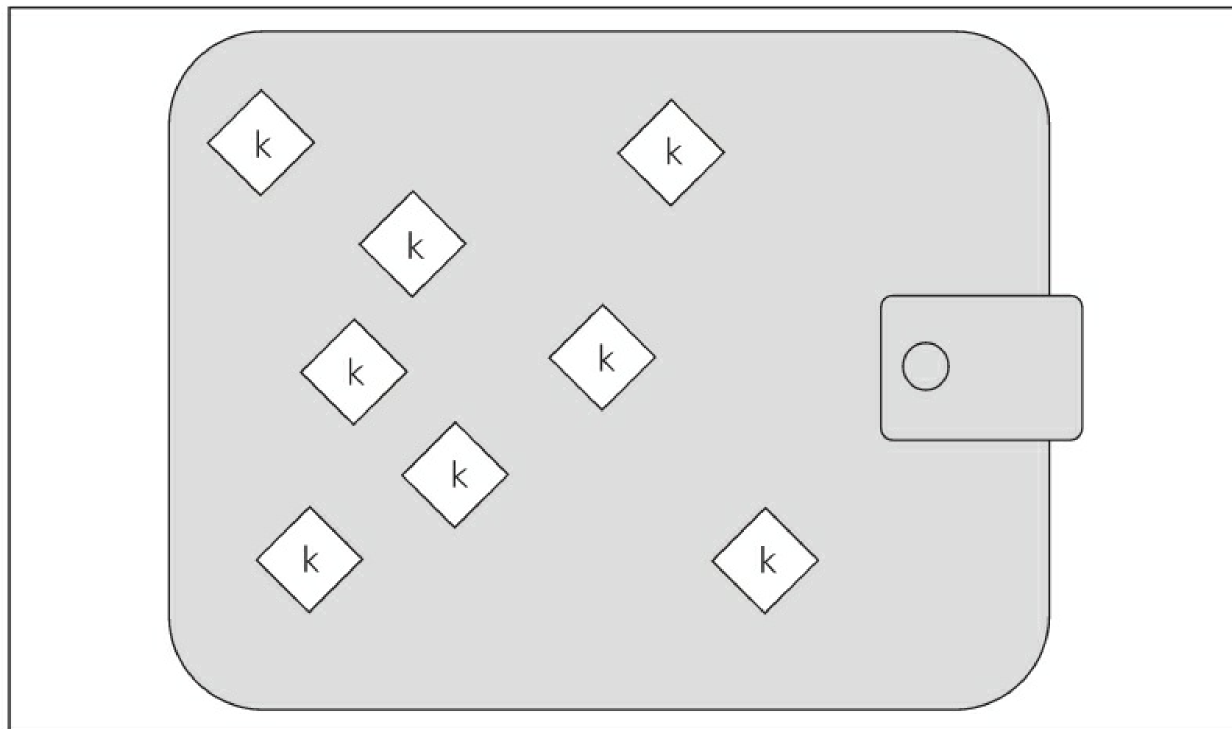


图4.8　Type-0非确定性（随机）钱包：随机生成密钥的集合

确定性（种子）钱包

确定性或者称为“种子”钱包是另一种钱包类型，它包含的私钥是通过使用单向哈希函数，从一个共同的种子衍生而来的。种子是一个随机生成的数字，它与其他诸如索引号或者“链码”［参见本章中“层次化确定性钱包（BIP0032/BIP0044）”］等组合并计算得出私钥。在确定性钱包中，只要使用种子就可恢复所有的衍生密钥，也就是说，只要在创建钱包时做个简单备份就够了。种子也一样可以在钱包导入或导出时使用，利用种子可以非常简单地将用户的所有密钥从一个钱包软件迁移到

另一个钱包软件。

助记码词汇表

助记码是一些英文单词序列，用于代表（编码）一个随机数字，这个随机数字就是用于创建确定性钱包的种子。这个单词序列足以重建种子，并根据种子重建钱包及所有派生而来的密钥。一个实现了确定性钱包（带助记码功能）的应用，在首次启动时将会向用户展示一个12到24个单词的序列。这个单词序列就是钱包的一个备份，可以用于在相同的或者兼容的应用中恢复并重建所有密钥。助记码词汇表使用户钱包备份变得极为简单，毕竟相对一串随机数字，这些词汇更容易阅读和转录。

助记码在比特币改进提案39（BIP0039）中首次被定义，目前还属于草案状态。需要注意的是，BIP0039仍是一个草案，而不是标准。尤其，还有一个不同的标准，使用了一套不同的词汇，在以太坊钱包（Electrum Wallet）中使用，并且其定义要早于BIP0039。BIP0039已被Trezor钱包和其他一些钱包软件使用，但是与以太坊的应用程序不兼容。

BIP0039按如下步骤定义助记码和种子。

1.创建一个128位到256位的随机序列（熵）。

2.创建随机序列的校验码，即随机序列的SHA256哈希值的前几位。

3.将校验码附加到随机序列之后。

4.将序列拆成11位长的小段，使用这些小段与一个预定义的包含2048个单词的词典做对应[注]。

5.生成12到24个单词作为助记码。

表4.5显示了熵的大小与助记码长度的关系。

表4.5　助记码：熵与词汇数量

熵（位数）	校验码（位数）	熵+校验码	词汇数量
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

助记码代表了128到256位的数字，使用密钥扩展函数PBKDF2，可以产生更长（512位）的种子，所产生的种子用于创建确定性钱包以及所有派生的密钥。

表4.6和表4.7显示了一些助记码及其所生成的种子。

表4.6　128位助记码和产生的种子

熵输入（128 位）	0c1e24e5917779d297e14d45f14e1a1a
助记码（12 个单词）	army van defense carry jealous true garbage claim echo media make crunch
种子（512 位）	3338a6d2ee71c7f28eb5b882159634cd46a898463e9d2d0980f8e80dfbba5b0fa0291e5fb888a599b44b93187be6ee3ab5fd3ead7dd646341b2cdb8d08d13bf7

表4.7　256位助记码和结果种子

熵输入（256 位）	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
助记码（24 个单词）	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
种子（512 位）	3972e432e99040f75ebe13a660110c3e29d131a2c808c7ee5f1631d0a977fcf473bee22fce540af281bf7cdeade0dd2c1c795bd02f1e4049e205a0158906c343

层次化确定性钱包（BIP0032/BIP0044）

开发确定性钱包的目的是实现从一个“种子”生成很多密钥。形式最先进的确定性钱包是层次化确定性钱包，或者叫**HD**钱包，在BIP0032标准中是这样被定义的。层次化确定性钱包所包含的密钥是一种树形结构，一个父密钥可以派生出一系列的子密钥，每个子密钥又可以派生出一系列孙密钥。以此类推，直到树的深度达到无穷大。树的结构如图4.9所示。

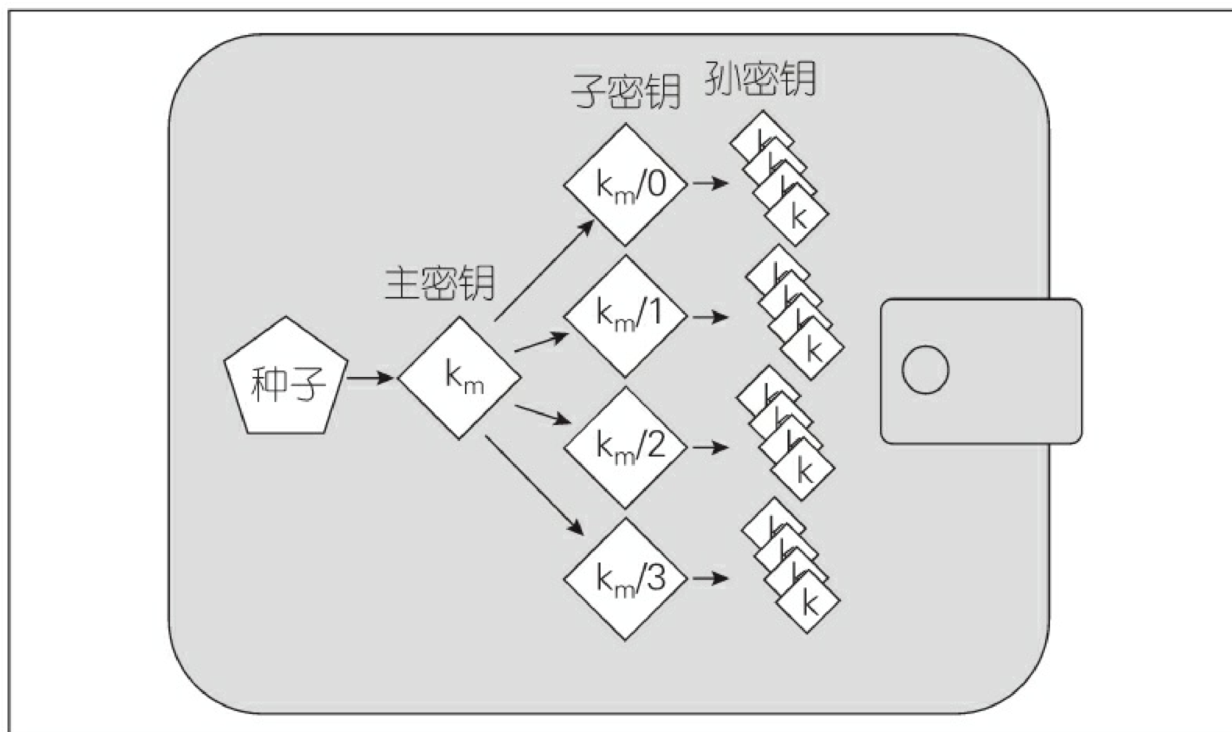


图4.9　Type-2层次化确定性钱包：从种子生成一棵密钥树

如果你自己开发一个比特币钱包软件，那么它必须是一个HD钱包，遵循BIP0032和BIP0044标准。

相对随机（非确定性）钱包，HD钱包有两大优势。第一，树结构可以用于表达额外的组织含义，比如，一个特定分支的子密钥用于接收来款交易，而另一个分支的子密钥用于支付交易的找零。不同分支的密钥同样可以用于公司财务设置，将不同的分支分配给不同的部门或子公司，用于特定用途或者会计账目。

第二，用户可以在不访问私钥的情况下创建一系列公钥。这使得HD钱包可以在不安全的服务器上使用，也可以针对每笔交易发放一个不同的公钥。公钥不需要提前预载或者派生，服务器也不需要保存用于花费资金的私钥。

从种子生成**HD**钱包

HD钱包从单一的根种子产生，它是一个128位、256位或者512位的随机数字。HD钱包中其他的一切东西均确定性地从这个根种子衍生而来，从种子起步重建整个兼容HD钱包而得以实现。同样地，包含成千上万密钥的钱包也更加易于备份、恢复、导出和导入，所有要做的仅仅是传输一个根种子。根种子最常见的表现形式为助记码单词序列，它使根种子的转录和保存更容易进行，在上一节“助记码词汇表”中我们已经详细描述过。

图4.10是HD钱包创建主密钥和主链码的过程。

根种子是HMAC-SHA512算法的输入，生成的哈希值用于创建主私钥（m）和主链码。通过主私钥（m）相应地生成一个主公钥（M），这个过程使用了我们之前介绍过的椭圆曲线乘法m×G。链码用于引入熵，在从父密钥创建子密钥的过程中需要用到，我们将在下节中讲述。

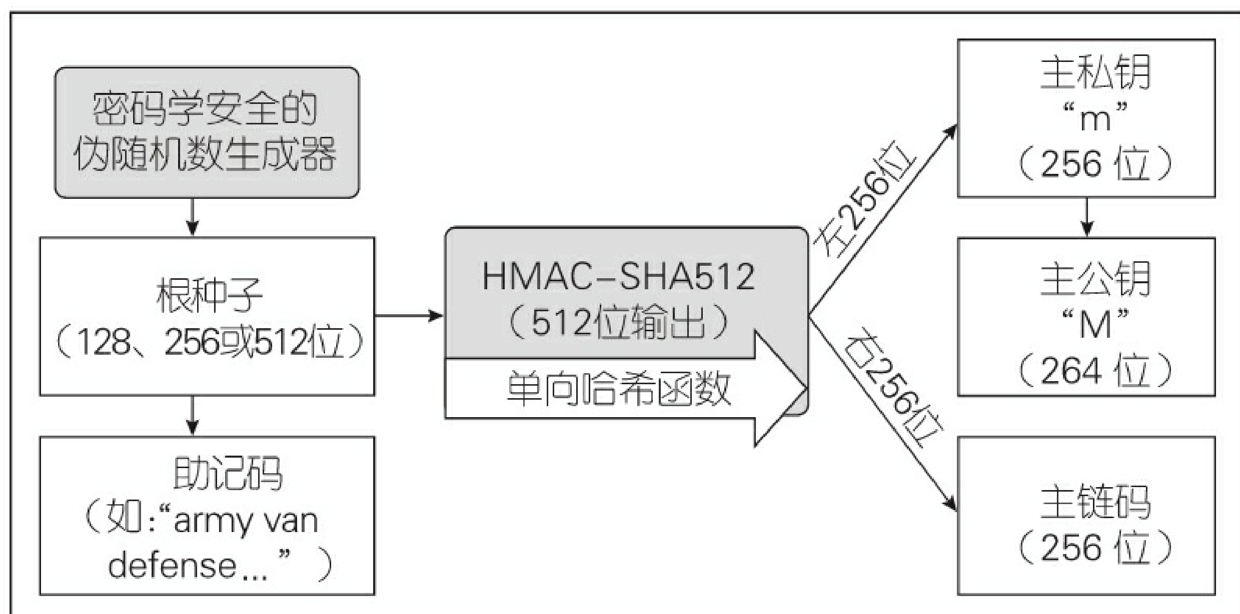


图4.10　从根种子创建主密钥和链码

子私钥的派生

层次化确定性钱包使用一个子密钥派生（CKD）函数从父密钥派生出子密钥。

子密钥派生函数基于单向哈希函数，它包括如下。

●一个父私钥或公钥（ECDSA非压缩密钥）。

●一个叫作链码的种子（256位）。

●一个索引号（32位）。

链码的作用是引入一个看似随机的数据到这个过程中，这样一来单凭索引就不足以派生出其他子密钥。因此，除非你也知道链码，否则只拥有一个子密钥是无法找到其同辈密钥的。初始链码种子（在树的根部）是由随机数生成的，但是子节点的链码是从父节点的链码派生而来的。

这三个元素组合并哈希生成子密钥。如图4.11所示。

父公钥，链码，索引号组合后，采用HMAC-SHA512算法进行哈希计算，生成512位的哈希。这个生成的哈希被分成两半。右边的256位哈希输出成为子节点的链码，左边的256位哈希与索引号被加入父私钥从而形成子私钥。在图4.11中，我们可以看到从父密钥生成第0个子密钥（索引号为0）的过程。

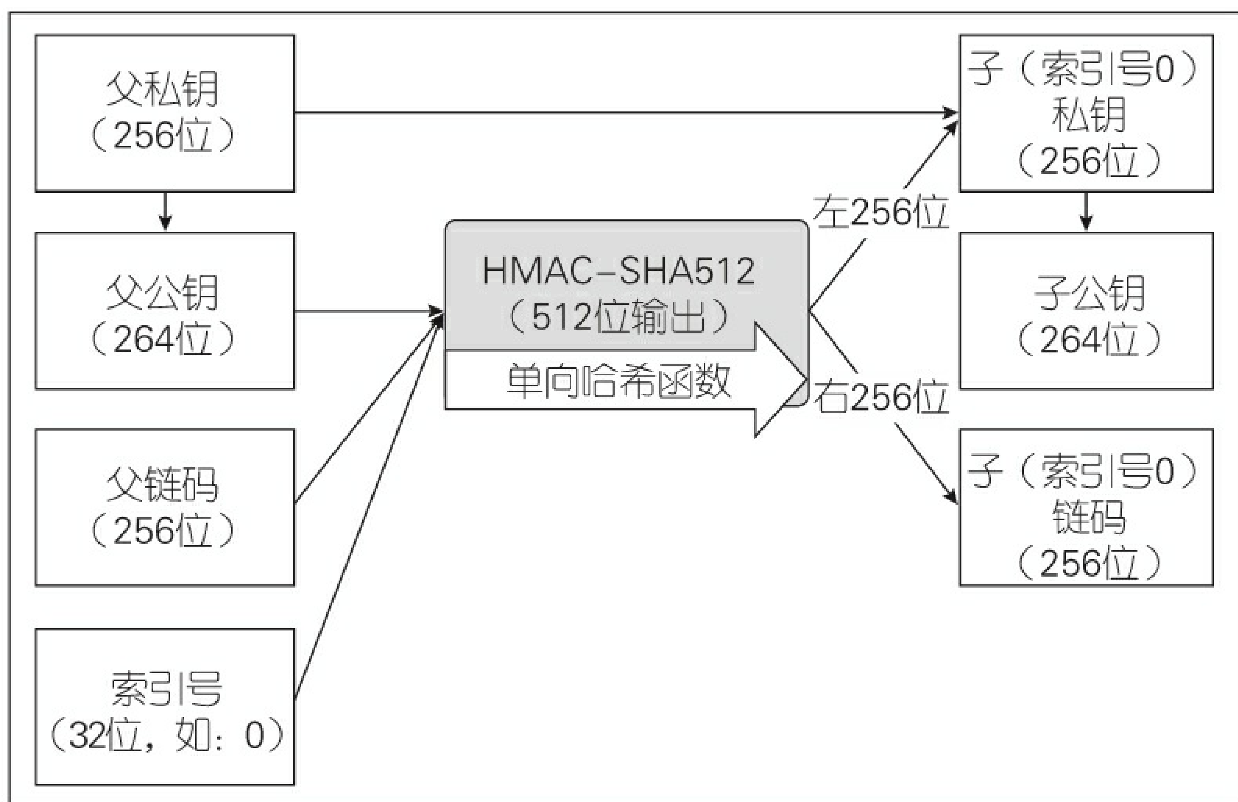


图4.11　扩展父私钥生成子私钥

改变索引号能够让我们扩展父密钥，并顺序生成其他子密钥，比如子密钥0、子密钥1、子密钥2等。每个父密钥最多可以产生20亿个子密钥。

每当从树上的当前位置下降一层，重复以上步骤，子密钥就成了父密钥，可以创建它们自己的子密钥，以此类推，可以产生无穷多的后代。

使用派生子密钥

子私钥与非确定性（随机）密钥是不可区分的。因为派生函数是一个单向函数，子密钥不能用于找寻其父密钥。子密钥同样不能用于找寻任何同辈密钥。如果你拥有第n个子密钥，就无法找到它的同辈，比如第n-1个子密钥，或者第n+1个子密钥，或者其他任何序列中的子密钥。只有父密钥和链码可用于派生所有子密钥。没有子链码，子密钥也无法

派生出任何孙密钥。你需要同时拥有子私钥和子链码才能开始建立一个新的分支，并派生孙密钥。

那么，子密钥本身可以拿来做什么呢？它可以用于产生一个公钥，一个比特币地址。然后你可以将其用于对发送到该地址上的未花费输出进行签名，从而解锁资金。

子私钥、相应的公钥及比特币地址，与随机生成的密钥、地址没有任何区别。事实上，它们构成的序列在生成它们的HD钱包以外是不可见的。一旦创建，它们与“普通”密钥并没有任何区别。

扩展密钥

就像之前看到的，密钥派生函数可以在树结构的任何层次上基于三个要素——密钥、链码、索引号，创建新的子密钥。两个至关重要的要素是密钥和链码，它们的组合叫作扩展密钥。名词“扩展密钥”也可以理解为“可扩展的密钥”，因为这样一个密钥可用于派生子密钥。

扩展密钥将一个256位密钥和256位链码连接成一个512位的序列，并进行存储和展现。有两种类型的扩展密钥：一个是扩展私钥，它是私钥和链码的组合，用于产生子私钥（并由此产生子公钥）；另一个是扩展公钥，它是公钥和链码的组合，可用于创建子公钥，就如本章“生成一个公钥”中所描述的。

将扩展公钥想象为HD钱包树状结构中一个分支的根节点，从分支的根节点出发，可以派生出分支下的其他节点。扩展私钥可以创建完整的分支，而扩展公钥只能创建分支中的公钥。

一个扩展密钥包括一个私钥或者公钥、链码。一个扩展密钥可以创建其子密钥，在树状结构中生成它自己的分支。分享一个扩展密钥可以授权对整个分支的访问权限。

扩展密钥采用Base58Check编码格式，以便在不同的BIP0032兼容钱包软件之间进行导入和导出。扩展密钥的Base58Check编码采用一个特殊的版本号，其前缀为“xprv”和“xpub”，以便于识别。由于扩展密钥是512或513位，它比我们之前看到的其他Base58Check编码的字符串要长得多。

下面是一个Base58Check编码的扩展私钥的例子。

```
xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CA-
WrUE9i6GoNMKUga5biW6Hx4tws2six3b9c
```

下面是其对应的扩展公钥，同样以Base58Check格式进行编码。

```
xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KE-
CeeMVKZBPLrtJunSDMstweyLXhRgPxdp14sk9tJPW9
```

子公钥派生

正如之前提到过的，层次化确定性钱包的一个很有用的特性就是可以从父公钥派生子公钥，而不需要其私钥。因此，我们可以使用两种方式派生子公钥：从子私钥计算得出，或者直接从其父公钥派生出来。

如此说来，一个扩展公钥可以用于派生其HD钱包结构的分支下的所有子公钥（只有子公钥）。

当服务器或者应用程序只有一个扩展公钥的副本而没有任何私钥时，利用这种快捷的公钥派生方式，可以创建非常安全的仅限公钥的部署。这种部署方式可以创建无穷多的公钥和比特币地址，但是不能花费任何发送到这些地址上的资金。同时，在其他更安全的服务器上，扩展私钥可以派生出与扩展公钥相对应的私钥，用于签署交易并花费资金。

这种解决方案下的一个典型应用场景是在一台电子商务应用的web服务器上安装扩展公钥。web服务器可以利用公钥派生函数创建一个全新的比特币地址用于每笔交易（比如，用于客户的购物车）。没有任何私钥部署在Web服务器上，这样就避免了私钥丢失的风险。若没有HD

钱包，唯一的解决途径就是在分开部署的安全服务器上创建成千上万个比特币地址，然后将它们预加载到电子商务服务器上。这种方式很麻烦，需要持续的维护工作，以避免电子商务服务器用尽那些密钥。

另外一个场景是用于冷存储或硬件钱包。在这种情景下，扩展私钥可以存储在一个纸钱包或者硬件设备（比如Trezor硬件钱包）上，而扩展公钥可以在线保存。用户可以随时创建“接收”地址，而私钥却可以安全地存储在线下。为了花费资金，用户可以在离线签名比特币客户端上使用扩展私钥，或者使用硬件钱包设备（比如Trezor）来签署交易。图4.12显示了扩展父公钥派生子公钥的机制。

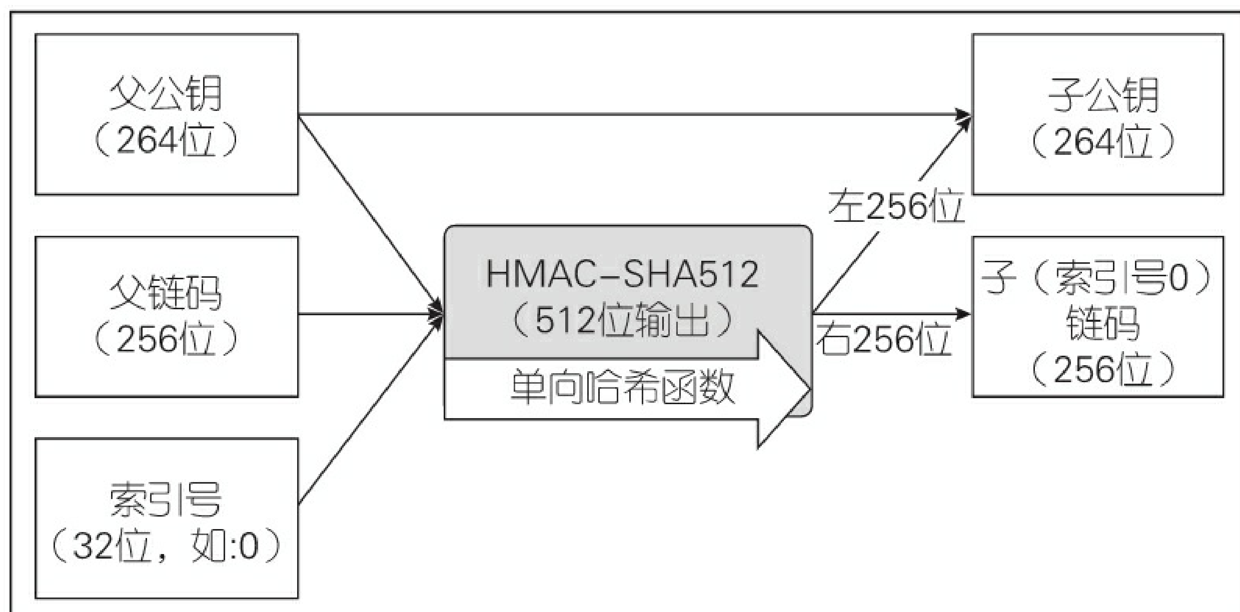


图4.12　扩展一个父公钥以创建子公钥

强化子密钥派生

从扩展公钥派生出公钥分支的能力是非常有用的，但是它有个与生俱来的潜在威胁。访问一个扩展公钥不会给予访问子私钥的权限，但是，因为扩展公钥包含链码，如果子私钥已知或者不小心泄露，那么就可以利用子私钥和链码派生出所有子私钥。一个泄露的子私钥，配合一个父链码，就将导致所有子节点的私钥的泄露。更糟糕的是，一个子私钥，连同父链码，可以推导出父私钥。

为防止这种威胁，HD钱包使用了一个替代的派生函数，叫作强化派生（**hardened derivation**），它断开了父公钥与子链码间的关系。强化派生函数利用父私钥来派生子链码，而不是父公钥。这样就创建了一个父/子序列间的“防火墙”，链码无法用来推断父辈私钥或同级私钥。派生函数看起来与普通子私钥派生一模一样，除了使用父私钥替代了父公钥，如图4.13所示。

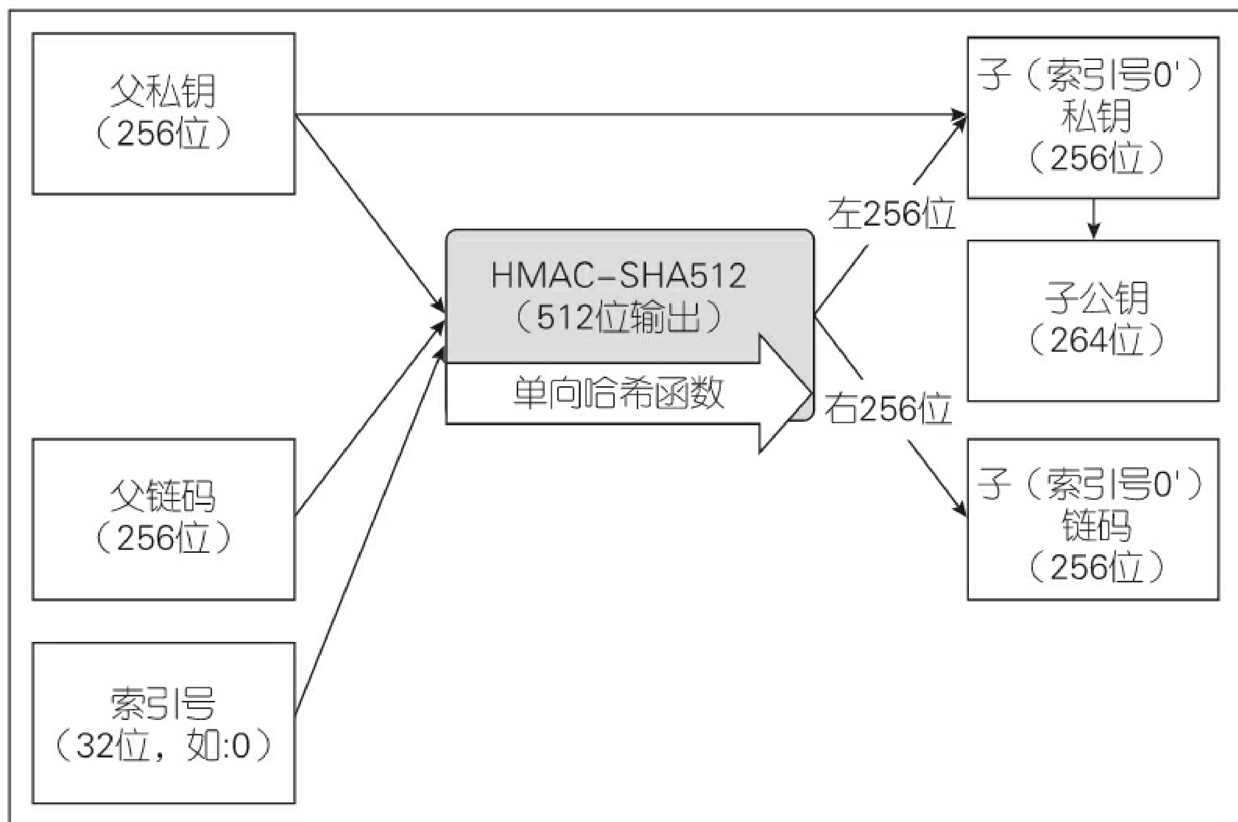


图4.13　子密钥的强化派生，省略了父公钥

当使用强化的私钥派生函数时，产生的子私钥和链码与普通的派生函数所产生的结果完全不同。其产生的密钥“分支”可以用于生成没有弱点的扩展公钥，因为它们包含的链码无法用于推导任何私钥。强化派生的引入，在扩展公钥所在层级与其上层级间形成了一个隔离。

简而言之，如果你想利用扩展公钥派生公钥分支的便利性，又不想使自己暴露在链码泄露的风险中，你需要从一个强化的父密钥来派生，

而不是普通父密钥。作为最佳实践，为防止主密钥泄露，主密钥的第一层子密钥总是通过强化派生函数派生而来。

普通派生和强化派生的索引号

派生函数中用到的索引号是一个32位的整数。为了易于区分从普通派生函数和强化派生函数派生的密钥，索引号被拆分成两个范围。介于0到2^{31}-1（0x0到0x7FFFFFFF）之间只用于普通派生，索引号介于2^{31}到2^{32}-1（0x80000000到0xFFFFFFFF）之间则只用于强化派生。这样，如果索引号小于2^{31}，意味着这是个普通子密钥，而如果索引号大于等于2^{31}，子密钥就是强化的。

为了使索引号易于阅读和显示，强化子密钥的索引号从0开始显示，但是带一个单引号。第一个普通子密钥显示为0，但是第一个强化子密钥（索引号为0x80000000）显示为0′。相应地，第二个强化密钥的索引号为0x80000001，显示为1′，以此类推。当你看到一个HD钱包的索引号为i′时，它与2^{31}+i是等价的。

HD钱包密钥标识符（路径）

HD钱包中的密钥使用“路径”命名规则进行标识，树结构的层与层之间采用“/”分隔符进行分隔。从主私钥派生而来的私钥开始于“m”，从主公钥派生而来的公钥开始于“M”。这样，主私钥的第一个子私钥表示为“m/0”，主公钥的第一个子公钥表示为“M/0”。第一个子私钥的第二个孙私钥表示为“m/0/1”，以此类推。

一个密钥的“血缘关系”是从右往左读的，一直到主密钥的位置。举例来说，标识符“m/x/y/z”描述一个密钥，它是密钥“m/x/y”的第z个子密钥，而“m/x/y”是“m/x”的第y个子密钥，“m/x”是m的第x个子密钥。具体例子如表4.8所示。

表4.8　HD钱包的路径举例

HD 路径	密钥描述
m/0	主私钥（m）的第一个（0）子私钥
m/0/0	第一个子私钥的第一个孙私钥
m/0’/0	第一个**强化**子私钥（m/0’）的第一个普通孙私钥
m/1/0	第二个子私钥（m/1）的第一个孙私钥
M/23/17/0/0	第 24 个子公钥的第 18 个孙公钥的第 1 个曾孙公钥的第 1 个曾曾孙公钥

HD钱包树状结构导航

HD钱包的树状结构提供了无比强大的弹性。每个父扩展密钥可以拥有40亿个子密钥：包括20亿个普通子密钥和20亿个强化子密钥。每个子密钥又拥有40亿个子密钥，以此类推。只要你愿意，这棵树可以任意扩展到无穷代。但是，正因为其弹性强，要对这棵“无限树”进行导航也变得极其困难。尤为困难的是在不同HD钱包间进行转移时，内部结构与分支及子分支间的映射关系也是无穷多的。

针对这一复杂性，两个比特币改进提案（BIPs）对HD钱包的树状结构提出了一些建议标准，提供了一些针对这个问题的解决方案。BIP0043建议第一个强化子索引作为一个特别的标识符，用于表示树结构的“目的”。基于BIP0043，HD钱包只能使用“层级1”的一个分支，通过定义索引号的目的，使用索引号来表示树的剩余部分的结构和命名空间。举例来说，一个HD钱包只使用分支m/i′/是为了标识一个特定目的，而这个目的由索引号“i”进行指定。

BIP0044扩展了上述规范，提出一种多账号结构作为BIP0043下索引号44′的“目的”。所有遵循BIP0044的结构仅使用树的一个分支：m/44′/。

BIP0044规定包含五个预定义层级的树状结构。

```
m / purpose' / coin_type' / account' / change / address_index
```

第一层“目的”（purpose）总是设置为44′。第二层“币类别”（coin_type）指定加密币类型，允许多币种HD钱包，每个币种在第二层下拥有自己的子树。目前已经定义了三种货币类型：比特币m/44′/0′；比特币测试网络（Bitcoin Testnet）m/44′/1′；莱特币（Litecoin）m/44′/2′。

第三层是“账户”（account），它允许用户将钱包分成几个逻辑独立的子账户，用作会计或组织机构用途。比如，一个HD钱包可能包含两个比特币“账户”：m/44′/0′/0′，m/44′/0′/1′。每个账户都是它们子树的根。

在第四层，“找零”（change），一个HD钱包有两个子树，一个用于创建接收地址，另一个用于创建找零地址。注意，不管上层是否使用强化派生，这层均使用普通派生。这是为了允许在这层树上可以导出扩展公钥用于不安全的环境中。可用的地址作为第四层的子密钥，是由HD钱包进行派生计算得来的，从而形成了第五层的“地址索引”（address_index）。比如：主账户中的第三个比特币支付接收地址就是M/44′/0′/0′/0/2。表4.9列出了更多的例子。

表4.9　BIP0044 HD钱包结构示例

HD 路径	密钥描述
M/44′/0′/0′/0/2	主账户上的第 3 个收款公钥
M/44′/0′/3′/1/14	第 4 个比特币账户上的第 15 个找零地址公钥
m/44′/2′/0′/0/1	莱特币主账户的第 2 个私钥，用于交易签名

使用**sx**工具包进行**HD**钱包试验

使用第3章介绍的命令行工具sx，你可以进行生成和扩展BIP0032确定性密钥的实验，并以不同格式展示它们：

```
$ sx hd-seed > m # create a new master private key from a seed and store in file "m"
$ cat m # show the master extended private key
xprv9s21ZrQH143K38iQ9Y5p6qoB8C75TE71NfpyQPdfGvzghDt39DHPFpovvtWZaR-
gY5uPwV7RpEgHs7cvdgfiSjLjjbuGKGcjRyU7RGGSS8Xa
$ cat m | sx hd-pub 0 # generate the M/0 extended public key
xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KE-
CeeMVKZBPLrtJunSDMstweyLXhRgPxdp14sk9tJPW9
$ cat m | sx hd-priv 0 # generate the m/0 extended private key
xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CA-
WrUE9i6GoNMKUga5biW6Hx4tws2six3b9c
$ cat m | sx hd-priv 0 | sx hd-to-wif # show the private key of m/0 as a WIF
L1pbvV86crAGoDzqmgY85xURkz3c435Z9nirMt52UbnGjYMzKBUN
$ cat m | sx hd-pub 0 | sx hd-to-address # show the bitcoin address of M/0
1CHCnCjgMNb6digimckNQ6TBVcTWBAmPHK
$ cat m | sx hd-priv 0 | sx hd-priv 12 --hard | sx hd-priv 4 # generate m/
0/12'/4
xprv9yL8ndfdPVeDWJenF18oiHguRUj8jHmVrqqD97YQHeTcR3LCeh53q5PXPkLsy2kRaqgwoS6YZ-
BLatRZRyUeAkRPe1kLR1P6Mn7jUrXFquUt
```

1. 2^{11}=2048，故每段均可以对应一个单词。——译者注

高级密钥和地址

在本节中，我们将观察密钥和地址的高级形式，比如加密私钥、脚本和多签名地址、荣耀地址，以及纸钱包。

加密私钥（BIP0038）

私钥必须一直保持私密。私钥的保密性要求在实践中是很难实现的，因为它与同等重要的安全目标——可用性，形成了直接冲突。当你需要保存私钥的备份以防丢失时，如何保证其私密性则更为困难。将私钥保存在钱包中，并通过密码加密会安全一些，但是这样钱包就需要进行备份。时不时地，用户可能还需要将密钥从一个钱包转移到另一个钱包——比如为了升级或者更换钱包软件。私钥备份可能也需要保存在纸上（参见本章中“纸钱包”），或者一个外置存储介质上，比如U盘。但是假如备份被盗或者丢失呢？这些冲突的安全目标，推动了一个私钥加密提案的诞生，它便携、方便，可以被不同的钱包和比特币客户端所理解。这个提案由比特币改进提案38号标准化，被称为BIP0038。

BIP0038提出了使用密码加密私钥，并进行Base58Check编码的标准，该标准保证密钥可以在备份介质上安全存储、在不同钱包间安全转移、在任何可能导致密钥泄露的场合保存。这个加密标准使用高级加密标准（AES），AES标准是美国国家标准技术研究所（NIST）创立的，广泛应用于商业及军事领域的数据加密。

BIP0038加密方案以一个比特币私钥作为输入，私钥通常以钱包导入格式（WIF）编码，是一个使用Base58Check编码，并带前缀“5”的字符串。另外，BIP0038提案还需要一个长密码——由多个单词或者包含

数字字母的复杂字符串。BIP0038提案的结构是一个Base58Check编码的加密私钥，以前缀“6P”开头。如果你看到一个密钥以“6P”开头，意味着它是一个加密的密钥，需要密码才能将其转换（解密）成为可以在钱包中使用的WIF格式的私钥（前缀为“5”）。很多钱包软件现在已经支持BIO0038加密私钥，会提示用户输入密码以便解密并导入。一些第三方应用，比如一款特别好用的基于浏览器的“Bit Address”（**http://bitaddress.org**）（在其“Wallet Details”选项卡中）就可用于解密BIP0038密钥。

BIP0038加密密钥最常用的案例是使用纸钱包，纸钱包用于在纸张上备份私钥。一旦用户选择了足够强大的密码，一个带BIP0038加密密钥的纸钱包将会非常安全，是比特币离线存储的极好选择（这也被称为“冷存储”）。

对表4.10中列出的加密密钥，可以使用bitaddress.org进行测试，了解如何通过输入密码得到解密密钥。

表4.10　BIP0038加密私钥示例

私钥（WIF）	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
密码	MyTestPassphrase
加密密钥（BIP0038）	6PRTHL6mWa48xSopbU1cKrVjpKbBZxcLRRCdctLJ3z5yxE87MobKoXdTsJ

支付到脚本哈希（P2SH）与多重签名地址

正如我们所了解的，传统比特币地址是以数字“1”开头的，从公钥计算得来，而公钥又是从私钥派生而来的。虽然任何人都可以往一

个“1”开头的比特币地址发送比特币，但是只有能提供与公钥哈希相对应的私钥，并进行签名的人，才能花费这笔比特币。

以“3”开头的比特币地址是支付到脚本哈希（Pay-To-Script Hash，简称P2SH）地址，有时也被错误地称作多重签名（multi-signature或multi-sig）地址。这种地址指定交易的受益人是一个脚本的哈希，而不是某个公钥的所有者。这项功能于2012年1月以BIP0016提案的形式引进，它为比特币地址本身提供了更多的功能，因此被广泛采纳。不像发送资金到“1”开头比特币地址［也被称为“支付到公钥哈希”（P2PKH）］的普通交易，发送资金到“3”开头地址，除了需要公钥哈希和作为所有权证明的私钥签名，还需要更多的东西。这些要求在地址创建时就在脚本中被指定，所有以这个地址作为目标的交易输入都会被这些要求所限制。一个发送到脚本哈希的地址是从一个交易脚本中创建的，它定义了谁可以花费这个交易输出［更详细的说明，参见第5章“支付到脚本哈希（P2SH）”］。编码一个支付到脚本哈希的地址，需要使用与创建比特币地址相同的双重哈希函数，只是它作用于脚本上而不是公钥上。

```
script hash = RIPEMD160(SHA256(script))
```

所产生的“脚本哈希”加上版本前缀“5”，并以Base58Check格式编码，得到一个以3开头的编码地址。这是一个P2SH地址的范例：

32M8ednmuyZ2zVbes4puqe44NZumgG92sM

P2SH不必与多重签名标准的交易相同。一个P2SH地址常常表示一个多重签名脚本，但它也可以表示其他类型交易的脚本编码。

多重签名地址和**P2SH**

目前，P2SH实现的功能大多是多重签名地址脚本，就像其名称所暗示的，底层脚本要求超过一个的签名来证明所有权并花费资金。比特

币多重签名功能的设计要求在N个密钥中，至少需要提供M个签名（被称为“阈值”），被称为M-of-M多重签名，这里M小于或等于N。举例来说，咖啡店老板鲍勃（第1章介绍的）可以使用一个多重签名地址，要求进行1-of-2签名，其中一个密钥来自鲍勃，另一个来自他妻子，以确保他们中的任何一个都可以签名花费被这个地址锁定的交易输出。这与传统银行提供的“联合账户”功能类似，联合账户允许夫妻中的任何一人签名即可使用资金。再例如，高佩什——那个为鲍勃设计网站的设计师，可以为其业务创建一个2-of-3签名地址，确保至少两个业务合作伙伴对交易签名后，才能花费发送到该地址的资金。

我们在第5章中将探索如何创建一个交易，以花费从P2SH（以及多重签名）地址接收到的资金。

荣耀地址

荣耀地址是一种包含人工可读信息的有效比特币地址。比方说1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33就是一个有效地址，它包含了英文单词“Love”，并作为Base-58字符的前面4位。创建一个荣耀地址往往需要尝试数十亿的候选私钥，直到找到一个能生成指定模式地址的私钥。虽然也有一些生成荣耀地址的优化算法，但是归根结底，其过程都是随机选择一个私钥，生成公钥，并由此生成地址，最后检查地址是否满足荣耀需求，重复这个过程直到找到匹配的选择项。

一旦找到一个与期望模式匹配的荣耀地址，与之相应的私钥就可用于花费比特币，这与其他地址完全一样。荣耀地址与普通地址相比没有任何区别。它们基于相同的椭圆曲线密码学（ECC）和安全哈希算法（SHA）。想通过荣耀地址查找私钥其难度也与普通地址一样。

在第1章中，我们介绍过尤金妮娅，一个菲律宾儿童慈善机构的负

责人。假设尤金妮娅要举办一个比特币的募捐活动，她想公布荣耀地址作为募捐地址。尤金妮娅创建了一个以“1Kids”开头的比特币地址，用于募捐宣传。下面我们来看看这个荣耀地址是如何创建的，以及这个地址对于尤金妮娅的慈善机构又意味着什么。

创建荣耀地址

首先，我们必须认识到比特币地址只是一个简单的数字，以Base58字符编码表示。为了找到“1Kids”开头的地址，可以看作从1Kids11111111111111111111111111111到1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzz中随便找到一个。则总共约有58^{29}（约等于1.4×10^{51}）个地址，都是以“1Kids”开头。表4.11显示了以1Kids开头的地址范围。

表4.11　以“1Kids”开头的荣耀地址范围

开始	1Kids11111111111111111111111111111
截止	1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzz

我们把模式“1Kids”看作一个数字，看一下在比特币地址里找到这个模式的频率（参看表4.12）。一台不带特殊硬件的台式电脑，每秒大概可以进行100000次密钥搜索。

表4.12　出现荣耀模式（1KidsCharity）的频率及在台式电脑上平均消耗时间

长度	模式	频率	平均搜索时间
1	1K	1/58	<1 毫秒
2	1Ki	1/3 364	50 毫秒
3	1Kid	$1/(195\times10^3)$	<2 秒
4	1Kids	$1/(11\times10^6)$	1 分钟
5	1KidsC	$1/(656\times10^6)$	1 小时
6	1KidsCh	$1/(38\times10^9)$	2 天
7	1KidsCha	$1/(2.2\times10^{12})$	3 ~4 个月
8	1KidsChar	$1/(128\times10^{12})$	13 ~18 年
9	1KidsChari	$1/(7\times10^{15})$	800 年
10	1KidsCharit	$1/(400\times10^{15})$	46 000 年
11	1KidsCharity	$1/(23\times10^{18})$	250 万年

就像你所看到的，尤金妮娅不可能马上创建“1KidsCharity”这样的地址，即使她能同时控制几千台计算机也难以做到。任意增加一个字符，难度就会增加58倍。超过7个字符的模式一般只能通过特定的硬件找到，比如定制的带有多块图形处理单元（GPUs）的电脑。这些电脑通常是比特币“矿机”再利用的产物，这些“矿机”用作挖矿已经无利可图，但是用来查找荣耀地址还是可行的。在GPU系统上查找荣耀地址要比使用通用的CPU快很多数量级。

查找荣耀地址的另外一种方式是外包给荣耀矿池，比如“Vanity Pool”矿池（**http://vanitypool.appspot.com**）。矿池是那些拥有GPU硬件的人所提供的一种服务，他们通过帮助别人查找荣耀地址挣取比特币。通过支付一笔很小的资金（0.01比特币，在写本书时大概相当于5美元），尤金妮娅就可以通过外包搜索得到一个7位数的荣耀地址，这个过程仅需几个小时，而如果自己用CPU计算的话，则需要花上几个月时

间。

创建荣耀地址是一种暴力破解过程：尝试一个随机密钥，检查它的地址是否与目标模式匹配，不断重复直到成功。例4-8列出了一个“荣耀地址矿工”的范例，它是一个基于C++语言的、用于查找荣耀地址的程序。例子使用了我们在第3章“替代客户端、库、工具集”中介绍过的libbitcoin库。

例**4-8**　荣耀地址矿工

```
#include <bitcoin/bitcoin.hpp>

// The string we are searching for
const std::string search = "1kid";

// Generate a random secret key. A random 32 bytes.
bc::ec_secret random_secret(std::default_random_engine& engine);
// Extract the Bitcoin address from an EC secret.
std::string bitcoin_address(const bc::ec_secret& secret);
// Case insensitive comparison with the search string.
bool match_found(const std::string& address);

int main()
{
    std::random_device random;
    std::default_random_engine engine(random());
    // Loop continuously...
```

```
    while (true)
    {
        // Generate a random secret.
        bc::ec_secret secret = random_secret(engine);
        // Get the address.
        std::string address = bitcoin_address(secret);
        // Does it match our search string? (1kid)
        if (match_found(address))
        {
            // Success!
            std::cout << "Found vanity address! " << address << std::endl;
            std::cout << "Secret: " << bc::encode_hex(secret) << std::endl;
            return 0;
        }
    }
    // Should never reach here!
    return 0;
}

bc::ec_secret random_secret(std::default_random_engine& engine)
{
    // Create new secret...
    bc::ec_secret secret;
    // Iterate through every byte setting a random value...
    for (uint8_t& byte: secret)
        byte = engine() % std::numeric_limits<uint8_t>::max();
    // Return result.
    return secret;
}

std::string bitcoin_address(const bc::ec_secret& secret)
{
    // Convert secret to pubkey...
    bc::ec_point pubkey = bc::secret_to_public_key(secret);
    // Finally create address.
    bc::payment_address payaddr;
    bc::set_public_key(payaddr, pubkey);
    // Return encoded form.
    return payaddr.encoded();
}

bool match_found(const std::string& address)
{
    auto addr_it = address.begin();
    // Loop through the search string comparing it to the lower case
    // character of the supplied address.
    for (auto it = search.begin(); it != search.end(); ++it, ++addr_it)
        if (*it != std::tolower(*addr_it))
            return false;
    // Reached end of search string, so address matches.
    return true;
}
```

示例代码必须使用C编译器进行编译，并与libbitcoin库连接（库必须预先安装在系统上）。运行示例时，只要运行vanity-miner++可执行程序，不需任何参数（参看例4-9），程序将尝试找到一个“1Kids”开头的地址。

例**4-9**　编译并运行**vanity-miner**示例程序

```
$ # Compile the code with g++
$ g++ -o vanity-miner vanity-miner.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the example
$ ./vanity-miner
Found vanity address! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT
Secret: 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f
$ # Run it again for a different result
$ ./vanity-miner
Found vanity address! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn
Secret: 7f65bbbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623
# Use "time" to see how long it takes to find a result
$ time ./vanity-miner
Found vanity address! 1KidPWhKgGRQWD5PP5TAnGfDyfWp5yceXM
Secret: 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349

real	0m8.868s
user	0m8.828s
sys	0m0.035s
```

示例代码需要花费数秒钟找到一个匹配3个字符模式“Kid”的结果，当使用Unix的time命令时我们就可以看到执行时间。你可以在源码中修改搜索模式，看看搜索4个甚至5个字符的模式需要多长时间。

荣耀地址的安全性

荣耀地址是一把双刃剑，既可以用于增强也可以削弱安全性。在提高安全性方面，一个与众不同的地址使入侵者难以将你的地址替换成他们的地址，以欺骗你的客户向他们付款。不幸的是，荣耀地址使得任何人都可以创建一个类似于随机地址的地址，或者另一个荣耀地址，以此欺骗你的客户。

尤金妮娅可以向愿意捐款的人宣布一个随机生成的地址（比如：1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy），或者她也可以生成一个以1Kids开头的易于识别的荣耀地址。

不管是哪种方式，均面临同样的威胁，在使用固定地址（而不是每个捐赠人一个独立的动态地址）的情况下，一旦有人侵入网站，并将这个地址替换为他自己的地址，捐赠人的资金将被转入他的账户。如果你通过不同的渠道发布捐赠地址，你的客户可以在确定支付前直观地检查这个地址，它是否与公布信息的网站、电子邮箱或者传单上所发布的地址相同。如果是一个随机地址，就像1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy这样，大部分客户可能会比对前面几个字符，比如1J7mdg，如果相同则认为地址是一样的。而那些有意盗取资金的入侵者，可以使用一个荣耀地址生成器，生成一个前面几位字符相同的替代地址，看起来与这个地址很接近，如表4.13所示的。

表4.13　生成与随机地址类似的荣耀地址

原始随机地址	1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
荣耀地址（4 个字符匹配）	1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy
荣耀地址（5 个字符匹配）	1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n
荣耀地址（6 个字符匹配）	1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX

那么荣耀地址能增强安全性吗？如果尤金妮娅生成一个荣耀地址1Kids33q44erFfpeXrmDSz7zEqG2FesZEN，客户很可能就会比对荣耀模式和其后的几个字符，比如“1Kids33”。这就使得攻击者必须至少生成6个字符（多2个字符），这将比尤金妮娅查找4位荣耀地址所花的时间多出3364倍（58×58）。最重要的，尤金妮娅付出的努力（或者向荣耀地址矿池交付的资金）使得攻击者不得不生成更长的荣耀地址。如果尤金妮娅缴费让荣耀地址矿池生成一个8个字符的荣耀地址，攻击者将不得

不创建10个字符长度的荣耀地址，这在普通电脑上已无法完成；如果采用荣耀地址挖矿机或者荣耀地址矿池，也是极为昂贵的。尤金妮娅可承受的费用到了攻击者这边就变得难以承受了，特别是如果潜在的回报无法覆盖生成荣耀地址所需要花费的费用时。

纸钱包

纸钱包就是将比特币私钥印制在纸张上。通常，出于方便起见，纸钱包也包含相应的比特币地址，不过这不是必需的，因为地址可以从私钥派生而来。纸钱包是一个非常高效的创建备份或离线比特币存储的手段，也被称为“冷存储”。作为一个备份机制，纸钱包可以提供足够的安全手段，防止因为计算机灾难造成的私钥丢失，比如硬盘失效、被盗或者意外删除。作为一种“冷存储”机制，如果纸钱包的密钥是离线生成，且从未在计算机系统上存储过，那么它们在对抗黑客、键盘记录，以及其他在线威胁方面更具安全性。

纸钱包可以有不同的形状、尺寸和外观设计，但是本质就是将密钥和地址打印在纸张上。表4.14显示了一个形式最简单的纸钱包。

表4.14　形式最简单的纸钱包——打印出来的比特币地址和私钥

公开地址	私钥（WIF 格式）
1424C2F4bC9JidNjjTUZ CbUxv6Sa1Mt62x	5J3mBbAH58CpQ3Y5RNJpUKPE62 SQ5tfcvU2JpbnkeyhfsYB1Jcn

使用一些工具，比如**bitaddress.org**上客户栏的JavaScript生成器，可以简便地生成纸钱包。这个页面包含生成私钥和纸钱包所必需的所有代码，即使与互联网完全断开，也不影响其使用。为了使用这个工具，要将HTML页面保存到本地硬盘或者外置U盘上，然后断开与互联网的连接，在浏览器中打开刚才保存的文件。更安全的做法是启动一个干净

的操作系统，比如一个CD-ROM启动的干净Linux系统，来使用这个工具。这个工具在离线状态下生成的任何密钥，均能通过USB线（不是无线）在本地打印机上打印出来。这样，创建的纸钱包密钥只在本地纸张上存在，而不会在任何在线系统上存储。将纸钱包存在一个防火的保险箱内，发送比特币到它们对应的地址，一个简单但是高效的“冷存储”方案就实现了。图4.14显示了一个通过bitaddress.org网站生成的纸钱包。



图4.14　通过bitaddress.org生成的简单纸钱包

简单纸钱包系统的缺点是打印的密钥容易被贼盯上。一个能接触到纸钱包的贼，既可以把那张纸偷走，也可以把密钥拍下来，从而控制被这些密钥所控制的比特币。更复杂的纸钱包存储系统采用BIP0038加密密钥。打印在纸钱包上的密钥受密码保护，而这个密码只在主人的脑袋里记着。没有密码，加密密钥毫无用处。不管怎样，纸钱包仍然比密码保护的钱包更安全，因为这些密钥从未在线，并且必须通过物理的方式从保险柜或者其他物理安全的存储设备上取得。图4.15显示了一个在bitaddress.org上生成的使用加密密钥（BIP0038）的纸钱包。



图4.15　通过bitaddress.org生成的加密钱包，密码是“test”

虽然你可以往一个纸钱包上多次存入资金，但是你必须一次性地用掉所有资金。这是因为在解锁和使用资金的过程中，你已经暴露了密钥，同样也因为当你花费的金额少于全部金额时，有些钱包应用会自动创建找零地址。解决这个问题的一种方案是，一次性地取出纸钱包地址上的所有余额，将剩余部分发送到另外一个新的纸钱包中。

纸钱包具有不同的设计方案和尺寸，也拥有不同的功能。有些设计方案用来作为礼物送给别人，因此设计了季节主题、比如圣诞主题、元旦主题等；另外一些则存在银行金库或者保险柜中，密钥需要通过某些方式进行隐藏，比如覆盖不透明的贴纸，折叠并使用防篡改黏合箔密封，等等。图4.16到图4.18是一系列带有安全保护和备份功能的不同的纸钱包。

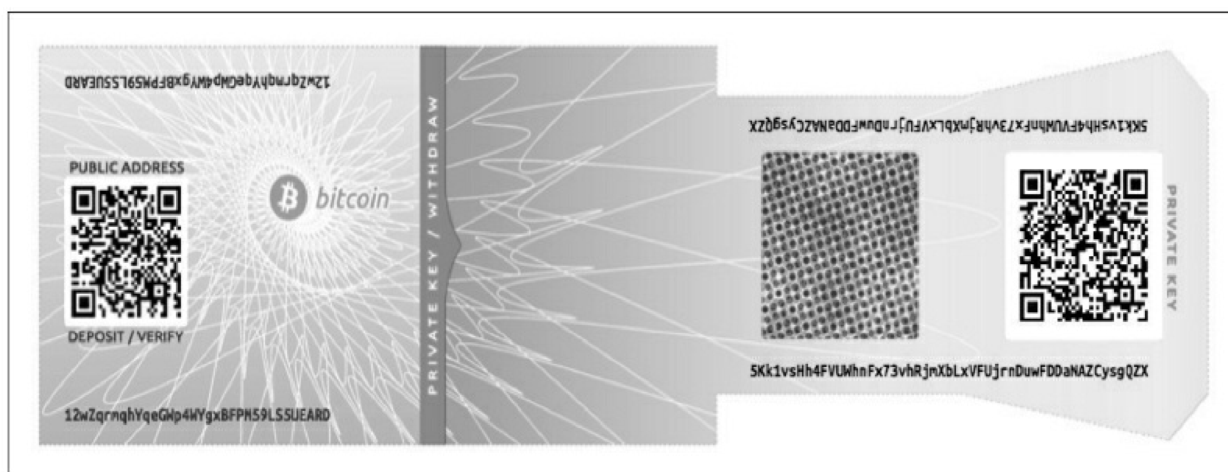


图4.16　通过bitcoinpaperwallet.com生成的纸钱包，密钥打印在折叠翼片上

另外一种设计方案，支持额外的密钥和地址的备份，形式上是一个附加的存根，类似票根，允许存储多个副本以达到防火防水及其他自然灾害的目的。



图4.17　通过bitcoinpaperwallet.com生成的纸钱包，密钥被密封隐藏



图4.18　具有额外密钥副本的纸钱包，印制在备份“存根”上

第5章 交易

交易是比特币系统中最重要的部分，其他的设计——比如确保交易能够顺利创建，得以在网络中传播、验证，最后加入全局交易账本（区块链）等技术，都是为交易服务的。交易是一个数据结构，用于对比特币系统中不同参与者间的价值传递进行编码。每个交易都是比特币区块链中的一条公开记录，而区块链则是一个全局的复式簿记账本。

在本章中，我们将研究交易的各种形态，比如它们包含什么内容，如何创建，如何验证，又将如何成为永久交易记录的一部分等。

交易的生命周期

交易的生命周期始于交易的创建，也被称为起源。然后交易被签上一个到多个签名，用于对交易中引用的资金进行授权。接着交易被传播到比特币网络，在那里，网络节点（参与者）对交易进行验证并继续传播出去，直到被网络中（几乎）所有节点获知。最后，交易被矿工节点确认，并被包含到一个交易区块中，记录到区块链上。

一旦被记录到区块链上，并被足够多的后续区块确认后，交易就成了比特币账本的永久组成部分，被所有参与者接受。交易中分配给新的所有者的资金，现在可以在新交易中使用了。就这样，所有者链条得以延伸，一个新交易的生命周期开始了。

创建交易

为理解交易的概念，可以将比特币交易想象成纸质支票。跟支票类似，交易也是一种表达资金转移意愿的工具。在提交执行前，它对金融系统是透明的；另外，交易的发起人不一定是对交易进行签名的人，这也跟支票一样。

任何人都可以在线或离线创建交易，甚至创建交易的人可以不是账户的有权签署人。举例来说，财务人员可能会处理一笔需经CEO签字的付款支票。相似地，财务人员也可以创建一笔比特币交易，然后经过CEO的数字签名使得交易生效。不同的是，支票需要关联一个账户作为其资金来源，而比特币交易是引用一个特定的前序交易作为其资金来源，而不是账户。

一旦交易被创建，它将被资金来源的所有者签名授权。如果交易的格式正确，签名合法，签名后的交易就被认定为有效，它包含了用于执行资金转移的所有必需信息。最后，有效交易必须送达比特币网络，并被传播出去，直到被一个矿工包含到公开账本（区块链）之中。

将交易广播到比特币网络

首先，交易需要送达比特币网络，这样才能被传播出去并被加入区块链。本质上，一个比特币交易只有300到400字节的数据，需要传播到成千上万的比特币节点。发送者无须委托广播交易的任何节点，只要它们使用超过一个节点确保交易能被传播出去就行。接收节点也不需要信任发送者或者确认它们的“身份”，因为交易已被签名并且不含任何机密信息，比如私钥、证书，交易可以通过任何方便的底层网络传输协议进行公开传播。信用卡交易由于包含敏感信息，只能通过加密网络传输数据，而比特币交易可以在任何网络上传输数据。只要交易送达能将其传播到比特币网络的节点，至于它是如何传递到第一个节点的，并不重要。

比特币交易可以通过某些不安全的网络传输到比特币网络中，比如Wi-Fi、蓝牙、近距离无线通信技术（NFC）、线性调频、条形码，甚至可以复制粘贴到一个web表单中。在极端情况下，比特币交易还可以通过无线电分组交换网、卫星中继、突发短波、扩频，或者防止检测和干扰的跳频的方式进行数据传输。比特币交易甚至也可以编码为一段表情符号（情感符），发布到公共论坛上，或者以文本消息或者Skype消息的形式发送。比特币将货币转换成一种数据结构，实际上已经无法阻止任何人创建或执行比特币交易。

交易在比特币网络中传播

一旦比特币交易发送到任意一个与比特币网络相连的节点，交易就将被这个节点进行验证。如果有效，这个节点会继续将其传播到其他相连的节点上，交易发起者也会同步接收到一个成功应答。如果交易被验证为无效，接收节点将拒绝交易，并返回一个拒绝交易的消息给交易发起者。

比特币网络是一个点对点网络，意味着每个节点均会与一些启动时通过点对点协议发现的节点相连。整个网络是一种松散连接的网状结构，没有固定的拓扑结构或其他结构，所有节点都是平等的。消息包括交易和区块，从一个节点传播到与之相连的节点上。新加入网络中任何节点的有效交易，会发送到三到四个相邻节点，每个相邻节点又再次将其发送到三到四个新的相邻节点，以此类推。利用这种方式，在短短几秒内，一个有效交易像以指数级扩散的波纹一样，在网络中迅速传播，直到到达所有连接着的节点。

比特币网络被设计成能够高效、弹性地在所有节点间传播交易和区块，同时它又能够有效防止攻击。为防止网络垃圾、拒绝服务攻击或者其他针对系统的恶意攻击，每个节点均在传播交易前进行独立验证，有缺陷的交易将无法传出节点。这个交易验证规则将在第8章“独立交易验证”中进行详细解释。

交易结构

一个交易就是一个数据结构，它的功能是将从资金源到目标的价值传递过程进行编码，其中资金源在交易中被称为输入，目标被称为输出。交易输入和输出与账户或者用户身份没有关联。相反，你应该把它们想象为比特币资金，即一笔比特币，被一个特定的密钥锁定，只有所有者，或者知道那个密钥的人才能进行解锁。一个交易包含一系列字段，如表5.1所示。

表5.1　一个交易的结构

大小	字段	描述
4 字节	版本（Version）	指定这个交易需要遵循的规则
1 ~9 字节（VarInt）	输入计数器（Input Counter）	有多少个输入
可变长度	输入（Inputs）	一个或多个交易输入
1 ~9 字节（VarInt）	输出计数器（Output Counter）	有多少个输出
可变长度	输出（Outputs）	一个或多个交易输出
4 字节	锁定时间（Locktime）	Unix 时间戳或区块号

交易锁定时间（Transaction Locktime）

锁定时间定义一个交易可以被加入区块链的最早时间。大多数交易此值设置为0，即立即执行。如果锁定时间非0，并且小于5亿，它被解释为区块高度，意思是交易不要被包含在指定高度以下的区块中。如果大于5亿，它是一个Unix时间戳（从1970年1月1日以来的秒数），意思是交易不要在这个时间前被加入区块链中。锁定时间的功能类似于支票的延期支付。

交易输出和输入

比特币交易的基本结构单元是未花费输出，或者被称为UTXO，UTXO是一个不可拆分的比特币结构，锁定一个特定的所有者，记录在区块链上，并被全网看作一个货币单元。比特币网络跟踪数以百万计的所有有效（未花费）UTXO。当一个用户接收到比特币，金额就以UTXO的形式记录在区块链上。这样，一个用户的比特币资金可能会以UTXO的形式分散存放在数百个交易和区块上。实际上，没有任何东西会去记录一个比特币地址或者账户的余额，只有分散的UTXO，锁定到特定的所有者。用户比特币账户余额的概念是钱包应用软件从传统应用中继承而来的。钱包软件通过扫描区块链，收集所有属于这个用户的UTXO，以此来统计用户的余额。

比特币中没有账户或余额，只有散布在区块链中的UTXO。

一个UTXO可以由任意倍的“聪”构成，就像美元可以被分割为小数点后两位的“分”一样。比特币能够分割到小数点后8位，被称作“聪”。虽然UTXO可以是任意金额，但是一旦创建，它就是不可分割的，就如同一枚硬币不能剖成两半一样。如果UTXO比交易所需要的金额大，它也需要一次性花完，超出的部分通过在交易中找零被索回。换句话说，如果你有20比特币的UTXO，需要支付1比特币，交易首先要将20比特币的UTXO全部花完，那么就需要创建两个输出：一个是支付1比特币给指定的接收人，另外一个则将19比特币返回到你的钱包。结果是，绝大部分交易都需要创建找零输出。

如果一个顾客需要买一份1.5美元的饮料，她在她的钱包里找一些钞票和硬币以凑够这笔费用。如果钱刚好够用，顾客会选择精确的零钱（比如1张1美元纸币和2枚25分硬币，或者6枚25分的硬币）付款，不凑

巧的话，她可能会掏出一张5美元的纸币。如果给了店员太多的钱，比方说5美元，她会收到3.5美元的零钱，这些零钱将放回到她的钱包供以后的交易使用。

类似地，比特币交易不管多大金额，都需要从用户的UTXO中进行创建。用户无法将UTXO拆成两半，就像不能把一张纸票撕成两半来用一样。用户的钱包应用自动从可用的UTXO中选取不同的金额，组合成大于或等于所需金额的交易。

在真实生活中，比特币应用会使用不同的策略来满足采购金额的要求：组合较小单位、找到精确的零钱，或者使用单一的大于交易金额的单元并进行找零。所有这些复杂的UTXO组合都是由用户钱包软件自动完成的，用户看不到具体过程。只有当用户以编程的方式自己从UTXO中创建原始交易，才需要关心这个选择的过程。

交易消费的UTXO叫作交易输入，交易创建的UTXO叫作交易输出。如此，比特币价值不停地从一个所有者转移到另一个所有者，形成一个消费和创建UTXO的交易链条。交易通过当前所有者的签名解锁并消费UTXO，通过将其锁定到新的所有者的方式创建新的UTXO。

对于输出输入链来说，也有一个例外，它是一种特殊类型的交易，叫作铸币（**coinbase**）交易，铸币交易是每个区块的第一笔交易。这笔交易是矿工“赢家”放进区块的，作为矿工挖到区块的奖励。这也就是比特币系统在挖矿过程中发行新币的过程。我们将在第8章介绍这部分内容。

哪个更优先？输入还是输出，先有鸡还是先有蛋？严格来说，输出应该是更早产生的，因为铸币交易产生了新的比特币，而这笔交易不需要输入，凭空就产生了输出。

交易输出

每个比特币交易都产生输出，输出将被记录在比特币账本上。除一种情况外［参见本章中“数据输出（OP_RETURN）”］，几乎所有这些输出都创建可使用的比特币，被称为UTXO，这些UTXO会被全网识别，并可被新的所有者在将来的交易中花费。向某人发送比特币就是创建一个UTXO并注册到他的地址上，随后他就可以花费这笔UTXO。

UTXO将被所有完全客户端，通过其维护在内存中的数据库的方式进行跟踪，这个数据库叫作**UTXO**集合或者**UTXO**池。新交易将从UTXO集合中消费（花费）一个或多个输出。

交易输出包含两部分：

●比特币金额，“聪”的任意倍数，“聪”是比特币的最小单位。

●锁定脚本，也被称为“受限”，通过指定花费输出必须符合某种条件，将这个金额锁定。

对于前面提到的锁定脚本，其使用的交易脚本语言将在本章“交易脚本和脚本语言”中详细讨论。表5.2显示一个交易输出的结构。

表5.2　交易输出结构

大小	字段	描述
8字节	数量	以聪为单位的比特币价值
1~9字节（VarInt）	锁定脚本大小（Locking-Script Size）	用字节表示的后续锁定脚本长度
可变长度	锁定脚本（Locking-Script）	定义花费输出所需条件的脚本

在例5-1中，我们利用blockchain.info的API来查找特定地址的UTXO。

例**5-1**　通过脚本访问**blockchain.info**的**API**，查找特定地址对应的**UTXO**

```
# get unspent outputs from blockchain API

import json
import requests

# example address
address = '1Dorian4RoXcnBv9hnQ4Y2C1an6NJ4UrjX'

# The API URL is https://blockchain.info/unspent?active=<address>
# It returns a JSON object with a list "unspent_outputs", containing UTXO, like
this:
#{      "unspent_outputs":[
#   {
#     "tx_hash":"ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167",
#     "tx_index":51919767,
#     "tx_output_n": 1,
#     "script":"76a9148c7e252f8d64b0b6e313985915110fcfefcf4a2d88ac",
#     "value": 8000000,
#     "value_hex": "7a1200",
#     "confirmations":28691
#   },
#...
#]}

resp = requests.get('https://blockchain.info/unspent?active=%s' % address)
utxo_set = json.loads(resp.text)["unspent_outputs"]

for utxo in utxo_set:
    print "%s:%d - %ld Satoshis" % (utxo['tx_hash'], utxo['tx_output_n'], utxo['val
ue'])
```

运行脚本，我们将看到一个列表，其格式类似：“交易ID:UTXO的索引号，以‘聪’为单位计算的价值”。锁定脚本在例5-2中的输出并未显示。

例**5-2**　**get-utxo.py**脚本执行结果

```
$ python get-utxo.py
ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167:1 - 8000000 Sa-
toshis
6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0 - 16050000 Sa-
toshis
74d788804e2aae10891d72753d1520da1206e6f4f20481cc1555b7f2cb44aca0:0 - 5000000 Sa-
toshis
b2affea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4:0 - 10000000 Sa-
toshis
...
```

花费条件（受限）

交易输出将特定金额（单位为“聪”）与特定的受限或者锁定脚本相关联，明确了花费这个金额必须满足的条件。在大多数情况下，锁定脚本将输出锁定到一个特定的比特币地址上，从而将这笔资金的所有权转移给新的所有者。当爱丽丝向鲍勃咖啡店支付一杯咖啡的比特币时，她的交易创建了一个0.015比特币的输出，锁定到咖啡店的比特币地址上。这个0.015比特币的输出记录在区块链上，成为UTXO集合的一部分，也意味着在鲍勃的钱包上，这笔输出已成为可使用余额的一部分。当鲍勃选择花费这笔余额时，他的交易将解开这个受限，通过提供包含私钥签名的解锁脚本对输出进行解锁。

交易输入

简而言之，交易输入是一个指向UTXO的指针。它们通过引用交易哈希和UTXO在区块链中的顺序号指向一个特定的UTXO。为了花费UTXO，交易输入需要包含解锁脚本以满足UTXO设置的花费条件。解锁脚本通常就是证明锁定脚本中比特币地址所有权的签名。

当用户进行支付时，钱包通过选择可用的UTXO创建一笔交易。举例来说，为了创建一笔0.015的支付交易，钱包应用可能会选择一个0.01的UTXO和一个0.005的UTXO进行组合，以汇总成交易所需金额。

在例5-3中，我们使用一个“贪婪”算法，选择可用UTXO来创建一个特定支付金额的交易。在例子中，可用UTXO以常量数组的形式提供，但在现实中，可用的UTXO通常需要通过RPC访问比特币核心或者其他第三方API，就像在例5-1中看到的那样。

例**5-3**　一个用来计算总共可发送多少比特币的脚本

```python
# Selects outputs from a UTXO list using a greedy algorithm.

from sys import argv

class OutputInfo:

    def __init__(self, tx_hash, tx_index, value):
        self.tx_hash = tx_hash
        self.tx_index = tx_index
        self.value = value

    def __repr__(self):
        return "<%s:%s with %s Satoshis>" % (self.tx_hash, self.tx_index,
                                              self.value)

# Select optimal outputs for a send from unspent outputs list.
# Returns output list and remaining change to be sent to
# a change address.
def select_outputs_greedy(unspent, min_value):
    # Fail if empty.
    if not unspent:
        return None
    # Partition into 2 lists.
    lessers = [utxo for utxo in unspent if utxo.value < min_value]
    greaters = [utxo for utxo in unspent if utxo.value >= min_value]
    key_func = lambda utxo: utxo.value
    if greaters:
        # Not-empty. Find the smallest greater.
        min_greater = min(greaters)
        change = min_greater.value - min_value
        return [min_greater], change
    # Not found in greaters. Try several lessers instead.
    # Rearrange them from biggest to smallest. We want to use the least
    # amount of inputs as possible.
    lessers.sort(key=key_func, reverse=True)
    result = []
    accum = 0
    for utxo in lessers:
        result.append(utxo)
```

```
            accum += utxo.value
            if accum >= min_value:
                change = accum - min_value
                return result, "Change: %d Satoshis" % change
        # No results found.
        return None, 0

def main():
    unspent = [
        OutputInfo("ebad
faa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167", 1,  8000000),
        OutputIn
fo("6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf", 0,
16050000),
        OutputInfo("b2af
fea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4", 0,  10000000),
        OutputIn
fo("7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1", 0,
25000000),
        OutputIn
fo("55ea01bd7e9afd3d3ab9790199e777d62a0709cf0725e80a7350fdb22d7b8ec6", 17,
5470541),
        OutputIn
fo("12b6a7934c1df821945ee9ee3b3326d07ca7a65fd6416ea44ce8c3db0c078c64", 0,
10000000),
        OutputIn
fo("7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818", 0,
16100000),
    ]

    if len(argv) > 1:
        target = long(argv[1])
    else:
        target = 55000000

    print "For transaction amount %d Satoshis (%f bitcoin) use: " % (target, target/
10.0**8)
    print select_outputs_greedy(unspent, target)

if __name__ == "__main__":
    main()
```

如果不带参数运行**select-utxo.py**脚本，它会尝试构建一个UTXO集合（包含找零）来支付55000000聪（0.55比特币）。如果提供一个目标支付金额作为参数，脚本将选择UTXO来创建指定金额的支付。在例5-4中，我们执行脚本，并尝试创建一个0.5比特币（50000000聪）的支

付。

例**5-4**　运行一个**select-utxo.py**脚本

```
$ python select-utxo.py 50000000
For transaction amount 50000000 Satoshis (0.500000 bitcoin) use:
([<7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1:0 with 25000000
Satoshis>, <7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818:0 with
16100000 Satoshis>,
<6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0 with 16050000
Satoshis>], 'Change: 7150000 Satoshis')
```

一旦UTXO选定后，钱包应用便开始创建包含每个UTXO签名的解锁脚本，使它们满足锁定脚本的条件，从而可以花费。钱包应用加入这些UTXO的引用和解锁脚本作为交易输入。表5.3显示了交易输入的结构。

表5.3　交易输入结构

大小	字段	描述
32 字节	交易哈希（Transaction Hash）	指向待花费 UTXO 的指针
4 字节	输出索引（Output Index）	UTXO 的编号，从 0 开始
1 ~9 字节（VarInt）	解锁脚本大小（Unlocking-Script Size）	紧跟的解锁脚本的长度
可变长度	解锁脚本（Unlocking-Script）	满足 UTXO 锁定脚本条件的解锁脚本
4 字节	序号（Sequence Number）	目前未被使用的交易替换功能，设置为 0xFFFFFFFF

序号（sequence number）用于覆盖早于交易锁定时间的交易，这是比特币暂时未启用的功能。大多数交易将这个值设定为整数的最大值（0xFFFFFFFF），它会被比特币网络忽略。如果交易有个非零锁定时间，交易输入中至少有个序号要小于0xFFFFFFFF，以使锁定时间生效。

交易费用

大多数交易包含交易费用，提供给为比特币网络安全做出贡献的矿工作为报酬。矿工挖矿、收集费用和奖励将在第8章详细讨论。本节主要研究交易费用是如何包含进典型交易的。大多数钱包软件会计算并自动包含交易费用。但是如果你使用程序创建交易，或者使用命令行界面，就必须手工计算并包含这笔费用。

通过在每笔交易中包含一小笔费用，可以形成令交易被加入下一区块的激励，也能成为对“垃圾”交易和系统滥用的反激励措施。矿工挖出新区块，将交易记录到区块链上，并收集交易费用。

交易费用基于交易大小进行计算，以千字节为单位，而不是基于交易价值计算。总的来说，交易费用是基于网络中的市场力量来设置的。矿工们基于不同的规则包括交易费用，对交易的优先级进行排序，在一定条件下，他们也免费处理交易。交易费用影响交易处理的优先级，也就是说，含有足够费用的交易更有可能被包含进最近的下一个区块，而费用不足或者没有费用的交易就可能被延迟，并遵循尽量处理的原则在后面的区块中被包含，或者干脆就得不到处理。交易费用不是必需的，没有费用的交易最终可能也会被处理；但是附加一定费用会提高处理的优先级。

随着时间推移，交易费用的计算方式，以及它对交易优先级的影响也在变化。最初，交易费用是固定的，是网络中的一个常量。慢慢地，费用结构逐渐放宽，以便让基于网络容量和交易数量的市场力量对其产生影响。当前最小交易费用固定为0.0001比特币或者每千字节0.1毫比特，这也是最近刚从1毫比特降到这个值的。大多数交易均小于1千字节，但是对于有多个输入和输出的交易，就会更大一些。在将来的比特币协议修订版中，钱包应用软件可能会统计分析近期交易的平均费用，从而计算合理的交易费用并附加到交易中。

当前矿工基于交易费用对交易优先级进行排序，并打包进区块的算法，将会在第8章中详细介绍。

添加费用到交易中

交易的数据结构中并没有费用字段。实际上，费用隐含在交易输入汇总和交易输出汇总的差值中。交易输入加总扣除所有输出后，剩余的金额就成为交易费用，最终被矿工收集走。

交易费用是隐含的，是输入减输出的差额。

$$\text{Fees} = \text{Sum(Inputs)} - \text{Sum(Outputs)}$$

费用是交易中很容易让人感到迷惑的因素，但也是一个必须弄懂的关键点。如果用户自己创建交易，就必须确保不会因为疏忽而使用太少的输入以此形成一笔很大的交易费用。也就是说，你必须计算所有的输入，必要时创建找零，否则你将向矿工贡献一笔巨额的小费！

举例来说，如果你使用一个20比特币的UTXO来创建一笔1比特币的支付交易，那么你必须包含一个19比特币的找零输出，以使资金回到你的钱包。否则，余下的19比特币将被认定为交易费用，将你的交易含进区块的矿工将收走这笔交易费用。虽然你的交易处理优先级别提高了，而矿工也会因为收到一大笔交易费用而高兴，但是这很可能并不是你所希望的。

如果在手工创建的交易中忘记添加找零输出，交易找零将变为交易费。“不要找零钱了！”可能不是你希望的。

我们再来看看爱丽丝购买咖啡的过程，可以观察到在实践中这个流程是如何工作的。爱丽丝希望花费0.015比特币购买一杯咖啡。为确保交易能快速得到处理，她想在交易中添加一些费用，比方说0.001比特币，这意味着交易总费用是0.016比特币。她的钱包软件必须找到一些UTXO，加起来余额要大于等于0.016比特币。当然，如有必要就创建找零。假设爱丽丝的钱包中有一笔0.2比特币的可用UTXO。交易需要耗尽

这个UTXO，并创建两个输出，一个是支付给鲍勃咖啡店0.015比特币，另一个是回到爱丽丝钱包的交易找零的0.184比特币，还剩0.001比特币未分配，就作为这笔交易隐含的费用。

我们来看另一个场景。尤金妮娅——菲律宾的儿童慈善机构负责人，已经完成一项为学校儿童采购课本的募捐。她从世界各地接收到几千笔的小额捐款，总共有50比特币。因此，她的钱包里充满了非常小额的UTXO。现在，她想从当地一个出版商那里采购几百本课本，使用比特币支付。

当尤金妮娅的钱包应用尝试创建一个大额的支付交易时，它首先需要从大量小额的可用UTXO集合中抽取合适的UTXO作为交易输入。这笔交易需要抽取超过100个小额UTXO作为输入，而只有一个交易输出，即付款给出版商。一个包含这么多输入的交易，其大小将超过1千字节，可能需要2000到3000字节。其结果是，交易费用必须高过网络最低费用0.0001比特币。

尤金妮娅的钱包应用会通过测算交易大小，并将其与每千字节的费用相乘，得到合适的交易费用。很多钱包软件会对大笔交易多付一定的费用，以确保交易能被及早处理。付出更高的交易费用不是因为尤金妮娅花费的钱更多，而是因为她的交易更复杂，规模也更大——交易费用与交易涉及的比特币价值无关。

交易链条和孤儿交易

正如我们所看到的，交易构成了一根链条，一个交易花费了上一笔交易（父交易）的输出，并创建新的输出以供后一笔交易（子交易）使用。有时，整根交易链条需要互相依赖，比方说父交易、子交易、孙交易是被同时创建的，用以满足一个复杂的交易流程要求，这个流程要求有效的子交易要先于父交易进行签名。举例来说，这就是和币（CoinJoin）交易所使用的技术，这种交易需要所有参与者同时加入，以保护隐私。

当一个交易链条在网络中传输时，它们不一定总是按照相同的顺序到达。有时候子交易可能早于父交易到达。这种情况下，看到子交易的节点会发现交易引用了一个未知的父交易。节点不会直接拒绝这笔交易，而是先将其放入一个临时的交易池，等待它的父交易到来；与此同时，节点继续向其他节点传播这笔交易。没有父交易的交易池被称为孤儿交易池（**orphan transaction pool**）。一旦父交易到达，所有引用了父交易的UTXO的子交易都将从池子里释放出来，重新进行递归验证。然后，整个交易链条就可以被放进交易池，等待被含进区块。交易链条的长度不受限制，允许任意数量的世代交易同时传输。将孤儿交易放进孤儿交易池的机制，可以保证原本有效的交易不会仅仅因为其父交易延时到达而被拒绝，不管到达的顺序如何，它们从属的交易链条最终都会依照正确的顺序重建起来。

存放在内存中的孤儿交易数量是有数量限制的，这样可以防止针对比特币节点发起的拒绝服务攻击。限制数量在比特币标准客户端的源码中定义为：MAX_ORPHAN_TRANSACTIONS。如果池中的交易数量超过MAX_ORPHAN_TRANSACTIONS，一个或多个交易就会被随机从池中排除出去，直到池中的交易数量小于限制数量。

交易脚本和脚本语言

比特币客户端通过执行脚本来验证交易。比特币脚本是一个类似Forth的脚本语言。不管是UTXO上的锁定脚本（受限），还是包含签名的解锁脚本，都是用这种脚本语言写的。当交易被验证时，每个输入上的解锁脚本都将与相应的锁定脚本一起执行，以查看是否符合花费条件。

如今，比特币网络处理的大部分交易都是类似“爱丽丝付给鲍勃”这种形式的，它们都基于叫作“支付给公钥哈希”的相同脚本。但是，使用脚本锁定输出、解锁输入，意味着通过使用编程语言，交易可以包含无限多的条件。比特币交易不仅限于“爱丽丝付给鲍勃”这种形式和模式。

这还只是这种脚本语言表达能力的“冰山一角”。在本节中，我们将演示比特币交易脚本语言的组成元素，展示它如何用于表达复杂的支付条件，以及这些条件是如何在解锁脚本中被满足的。

比特币交易验证不是基于静态模式，而是通过执行脚本实现的。这种脚本语言允许表达几乎接近无限的条件。这也是比特币为何具备“编程货币”能力的原因所在。

脚本创建（锁定+解锁）

比特币的交易验证引擎依赖两种类型的脚本：一个是锁定脚本，一个是解锁脚本。

锁定脚本是放置在输出上的一个受限，它设定条件，只有满足这些

条件，输出才能在未来被花费掉。由于锁定脚本通常包含公钥或者比特币地址，所以过去它曾被称为脚本公钥（**scriptPubKey**）。在本书中我们称其为“锁定脚本”，更能体现这个脚本技术在更多领域应用的可能性。在大多数比特币应用中，所谓的锁定脚本在程序源码中一般体现为“scriptPubKey”。

解锁脚本是“解决”或者满足一个输出上的锁定脚本设置的条件，从而允许输出被重新使用。解锁脚本是每个交易输入的一部分。大多数情况下，它们包含一个数字签名，这个签名由用户的钱包应用根据私钥创建。过去，解锁脚本被称为脚本签名（**scriptSig**），因为它通常包含数字签名。在大多数比特币应用中，源码一般把解锁脚本写成scriptSig。在本书中，我们称其为“解锁脚本”，以体现更广泛的锁定脚本要求，因为不是所有锁定脚本都包含签名。

每个比特币客户端都通过同时执行锁定和解锁脚本来验证交易。对于交易中的每个输入，验证软件首先检索被引用的UTXO。这个UTXO包含锁定脚本，它定义了花费输出所需要满足的条件。验证软件随后读取输入中包含的用于尝试花费UTXO的解锁脚本，接下来验证软件将同时执行这两个脚本。

在早期的比特币客户端中，解锁和锁定脚本是被连接起来并顺序执行的。出于安全考虑，这种情况在2010年被改变了，因为当时发现存在漏洞，一个允许非法的解锁脚本推送数据入栈，并污染锁定脚本。在现在的实现中，脚本是分开执行的，利用堆栈在两个脚本间传递数据，接下来我们将进行具体说明。

首先，利用堆栈执行引擎运行解锁脚本。解锁脚本成功运行后［比如，没有“悬空”（dangling）操作符］，主栈（不是替代栈）将被复制，接着锁定脚本被执行。如果利用从解锁脚本堆栈上复制来的数据，执行锁定脚本的结果为“真”（TRUE），那么解锁脚本就成功满足了锁定脚本设定的条件，从而输入拥有花费这笔UTXO的有效授权。如果组

合脚本执行完后存在任何非真的结果，则输入是无效的，因为它无法满足设置在UTXO上的花费条件。请注意UTXO是永久记录在区块链上的，因此它不会因这笔交易失败而变化或受到影响。只有一笔有效的、能正确满足UTXO使用条件的交易，才会导致这笔UTXO被标注为“已花费”，并从可用（未花费）UTXO集合中被移除。

图5.1是一个最普通的比特币交易（支付到公钥哈希）的解锁和锁定脚本的例子，显示了在脚本验证前将解锁和锁定脚本连接形成组合脚本的结果。

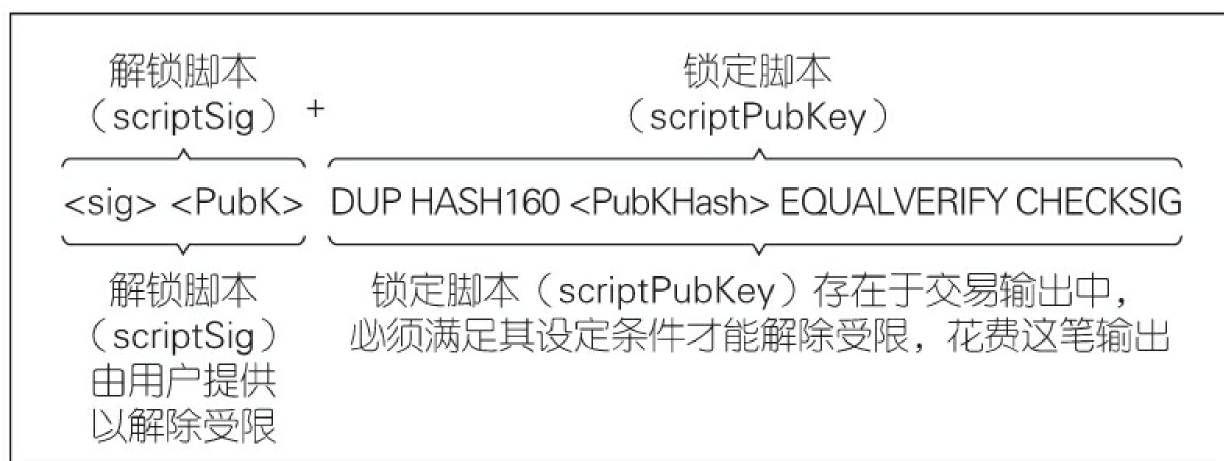


图5.1　组合scriptSig和scriptPubKey来评估一个交易脚本

脚本语言

比特币交易脚本语言，叫作脚本，是一个与Forth类似的逆波兰式表示的基于堆栈的执行语言。如果听起来感觉乱七八糟，可能是因为你没有学习过20世纪60年代的编程语言。脚本是一种非常简单的语言，只能执行限定的功能并且只能在一些特定的硬件上执行，它就像嵌入式设备，比如手持计算器，一样简单。它只要求最低的处理能力，也不能像其他现代编程语言一样可以做很多有意思的事情。在可编程货币的情境下，它其实是一种特别设计的安全特性。

比特币的脚本语言被称作基于堆栈的语言，因为它使用了一种叫作堆栈的数据结构。堆栈是一种非常简单的数据结构，你可以将它看作一堆卡片。堆栈只运行两种操作，入栈（push）和出栈（pop）。入栈是将一个项目添加到栈的顶部。出栈则从栈顶部移除一个项目。

脚本语言通过从左到右处理每个项目来执行脚本。数字（数据常量）被压入栈中。操作符将一个或多个参数压入栈中，或者从栈中移除，操作它们，并有可能将结果压入栈中。例如，OP_ADD从栈中移出两个项目，把它们相加，然后将求和结果压入栈中。

条件操作符评估一个条件，产生一个真（TRUE）或假（FALSE）的布尔结果。比如，OP_EQUAL从堆栈中移出两个项目，如果两个项目相等，则把TRUE（TRUE以数字1代表）压入栈中，如果不相等，则压入FALSE（用数字0表示）。比特币交易脚本通常都会包含条件操作符，因此可以产生TRUE的结果来表示一个有效交易。

在图5.2中，脚本23 OP_ADD 5 OP_EQUAL演示了算术加法操作符OP_ADD，将两个数相加，并把结果压入栈中；接着，条件操作符OP_EQUAL检查求和结果是否等于5。为了更加简洁，“OP_”前缀在后面的操作示例中将被省略。

以下是一个稍微复杂的脚本，用于计算2+7-3+1。注意，当脚本在一行中包含多个操作符时，堆栈允许一个操作符的结果被下一个操作符使用。

```
2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL
```

请使用铅笔和纸张验证一下前面的脚本。当脚本执行完毕，你会发现堆栈中只剩一个TRUE值。

虽然大多数锁定脚本都指向比特币地址或者公钥，从而要求证明所有者花费这笔资金的权限，但实际上，脚本并不要求一定要那么复杂。

任何锁定和解锁脚本的组合，只要能够得到一个为“真”的结果，就是有效的。前面用于说明脚本语言的例子中，简单的算术运算也是一个有效的锁定脚本，可用于锁定一笔交易输出。

使用算术运算例子的一部分作为锁定脚本。

```
3 OP_ADD 5 OP_EQUAL
```

只要交易包含这样一个解锁脚本，以上条件就能得到满足。

```
2
```

验证软件将上述锁定和解锁脚本进行组合，形成如下脚本。

```
2 3 OP_ADD 5 OP_EQUAL
```

我们可以在图5.2的操作范例中看到，当这个脚本被执行后，结果是OP_TRUE，使得交易有效。不仅这个交易输出锁定脚本有效，只要有一点算术技巧，知道数字2能满足算术运算结果，任何人都能够花费这笔UTXO。

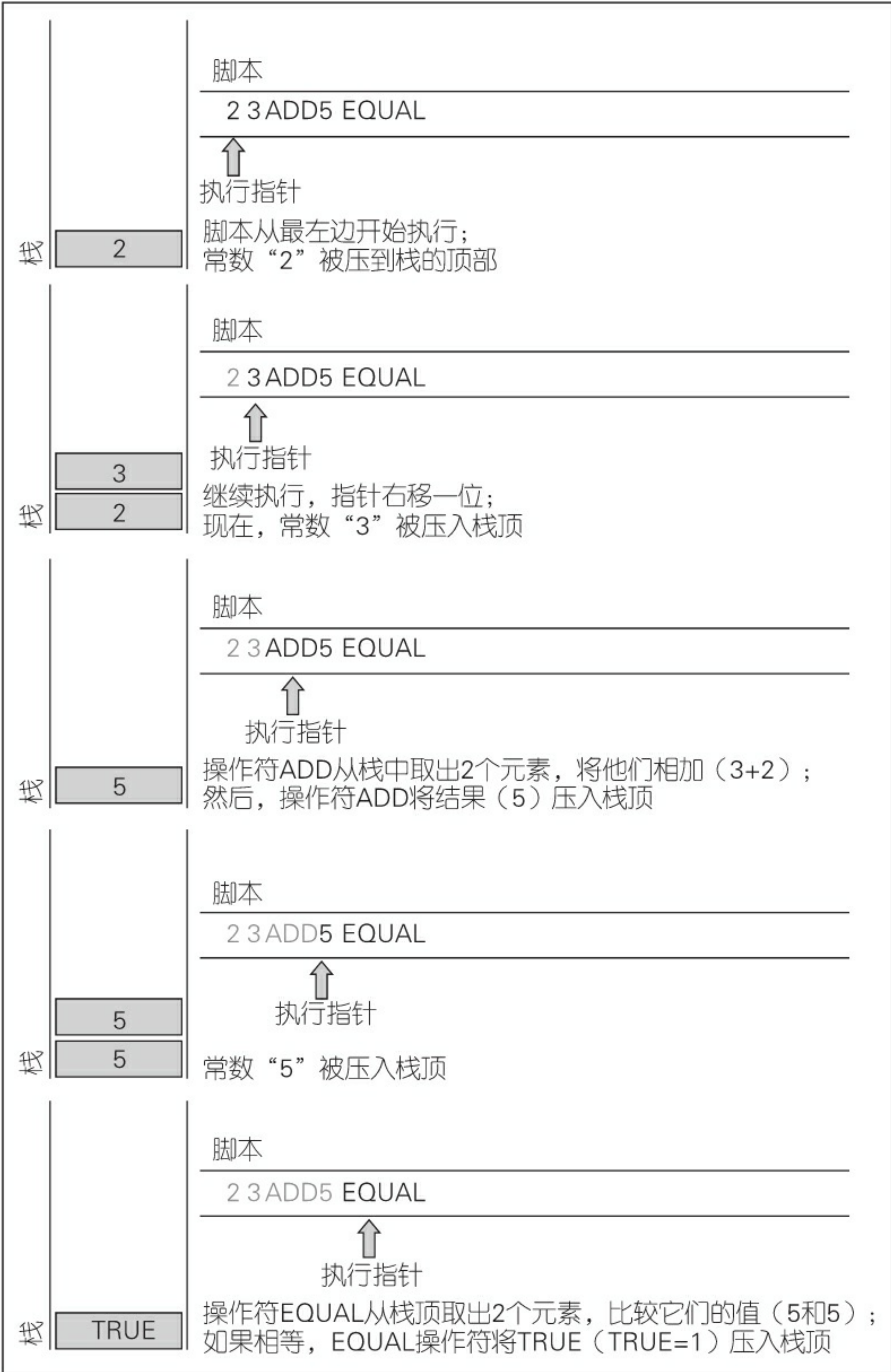

脚本
2 3 ADD5 EQUAL
执行指针
栈
2
脚本从最左边开始执行；
常数“2”被压到栈的顶部
脚本
2 3 ADD5 EQUAL
执行指针
3
2
栈
继续执行，指针右移一位；
现在，常数“3”被压入栈顶
脚本
2 3 ADD5 EQUAL
执行指针
栈
5
操作符ADD从栈中取出2个元素，将他们相加（3+2）；
然后，操作符ADD将结果（5）压入栈顶
脚本
2 3 ADD5 EQUAL
执行指针
5
5
栈
常数“5”被压入栈顶
脚本
2 3 ADD5 EQUAL
执行指针
栈
TRUE
操作符EQUAL从栈顶取出2个元素，比较它们的值（5和5）；
如果相等，EQUAL操作符将TRUE（TRUE=1）压入栈顶

图5.2　比特币脚本简单数学运算的验证过程

如果堆栈顶部的结果是TRUE（用{0×01}表示）、任何非零值或者脚本运行后堆栈为空，则交易有效；如果堆栈顶部的值是FALSE（一个零长度空值，记为{}），或者脚本执行被一个操作符显式终止，比如OP_VERIFY，OP_RETURN，或者一个条件终止符，如OP_ENDIF，则交易无效。详细情况见附录A。

图灵不完备

比特币交易脚本语言包含很多操作符，但是特意在一方面进行了限制——没有循环，也没有条件控制以外的复杂流程控制能力。这使得这种语言不是图灵完备的，意味着脚本的复杂性有限，执行时间也可以预测。脚本不是通用语言。这些限制条件确保该语言不能用于创建无限循环或者其他形式的“逻辑炸弹”，从而避免这些伎俩以某种方式嵌入到交易中，并导致对比特币网络产生拒绝服务攻击。记住，任何交易均会被网络上的所有完全节点验证。一个限定功能的语言能够防止交易验证机制被当作弱点利用。

无状态验证

比特币交易脚本语言是无状态的，在脚本执行前没有状态，执行后也不会保存状态。如此，所有需要被脚本执行的信息必须包含在脚本当中。可以预见的是，脚本在所有系统中都会以同样的方式运行。如果你的系统验证了脚本，可以确定其他比特币网络中的系统也一样能够验证这个脚本。也就是说，一个有效的交易对任何人都是同样有效的，而且任何人都知道这点。这种可预测结果的特性是比特币系统的一个重要特

点。

标准交易

在最初几年的比特币版本中，开发者对可被标准客户端处理的脚本设定了一些限制。这些限制被编码进一个函数，isStandard（），它定义了5种类型的“标准”交易。这些限制是临时性的，可能在你阅读本书时已经被移除了。限制取消之前，标准客户端仅能接受这5种标准类型的交易脚本。现实中，实际上大多数矿工都是运行标准客户端。虽然创建非标准的，即不属于5种标准脚本类型的交易是允许的，但是你首先要找到愿意不遵守这些限制的矿工，并将你的交易含进区块。

检查比特币核心客户端（标准程序）的源码，查看一下目前哪些交易脚本是被允许的。

5个标准类型的交易脚本包括：支付到公钥哈希（P2PKH）、公钥、多重签名（限定最多15个密钥）、支付到脚本哈希（P2SH），以及数据输出（OP_RETURN），这些将在接下来的几小节中详细介绍。

支付到公钥哈希（P2PKH）

绝大多数在比特币网络上处理的交易都是P2PKH交易。这种交易包含一个锁定脚本，锁定脚本通过公钥哈希（通常被称为比特币地址）阻碍了交易输出。对比特币地址进行支付的交易包含一个P2PKH脚本。一个被P2PKH脚本锁定的输出，可以通过提供公钥及与之对应的私钥签署的数字签名进行解锁（花费）。

让我们再来看看爱丽丝向鲍勃咖啡店发起支付的例子。爱丽丝发起一笔支付交易，向咖啡店的比特币地址支付0.015比特币。交易的输出

应该会有一个类似这样的锁定脚本。

```
OP_DUP OP_HASH160 <Cafe Public Key Hash> OP_EQUAL OP_CHECKSIG
```

咖啡店的公钥哈希等同于它的比特币地址，只是没有进行Base58Check编码。大多数应用以十六进制编码形式显示公钥哈希，而不是我们熟悉的Base58Check格式并以“1”开头的比特币地址。

以上的锁定脚本，可以被一个这样形式的解锁脚本满足。

```
<Cafe Signature> <Cafe Public Key>
```

将两个脚本组合成下列验证脚本。

```
<Cafe Signature> <Cafe Public Key> OP_DUP OP_HASH160
<Cafe Public Key Hash> OP_EQUAL OP_CHECKSIG
```

当且仅当解锁脚本匹配锁定脚本设置的条件时，这个组合脚本的执行结果才会为“真”（TRUE）。换句话说，当解锁脚本拥有咖啡店私钥的有效签名，而这个私钥又与设置为受限的公钥哈希相对应时，脚本执行结果为真（TRUE）。

图5.3和5.4（分两部分）一步步展示了组合交易执行的过程，这将证明交易的有效性。

支付到公钥（Pay-to-Public-Key）

相对支付到公钥哈希，支付到公钥是一个较为简单的比特币支付形式。这种形式下，公钥本身被存储在锁定脚本中，而不是像P2PKH一样只存储公钥哈希，哈希值要比公钥短得多。支付到公钥哈希是中本聪发明的，是为了让比特币地址变得更短，便于使用。支付到公钥现在一般只会在铸币交易中看到，它们由老版本的挖矿软件创建，这些软件一直没有更新到可以使用P2PKH的版本。

一个支付到公钥的锁定脚本看起来就像这样。

```
<Public Key A> OP_CHECKSIG
```

相应的解锁脚本仅需简单地提供一个签名。

```
<Signature from Private Key A>
```

组合脚本，用于交易验证软件验证。

```
<Signature from Private Key A> <Public Key A> OP_CHECKSIG
```

这个脚本是对CHECKSIG操作符的简单调用，用来验证签名是否由正确的密钥产生。如果正确，就返回结果“真”到堆栈中。

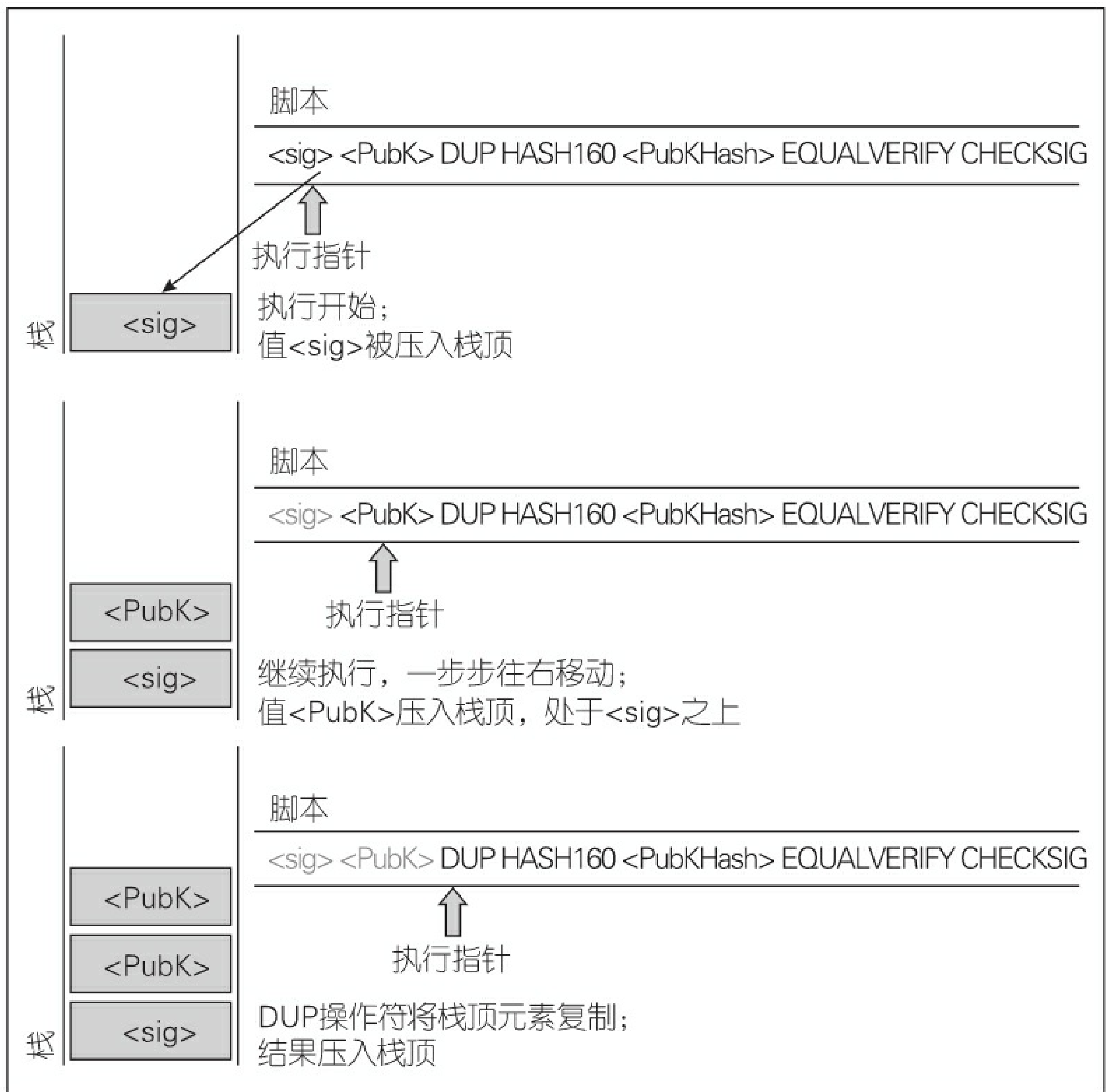


图5.3　评估一个P2PKH交易的脚本（第1部分）

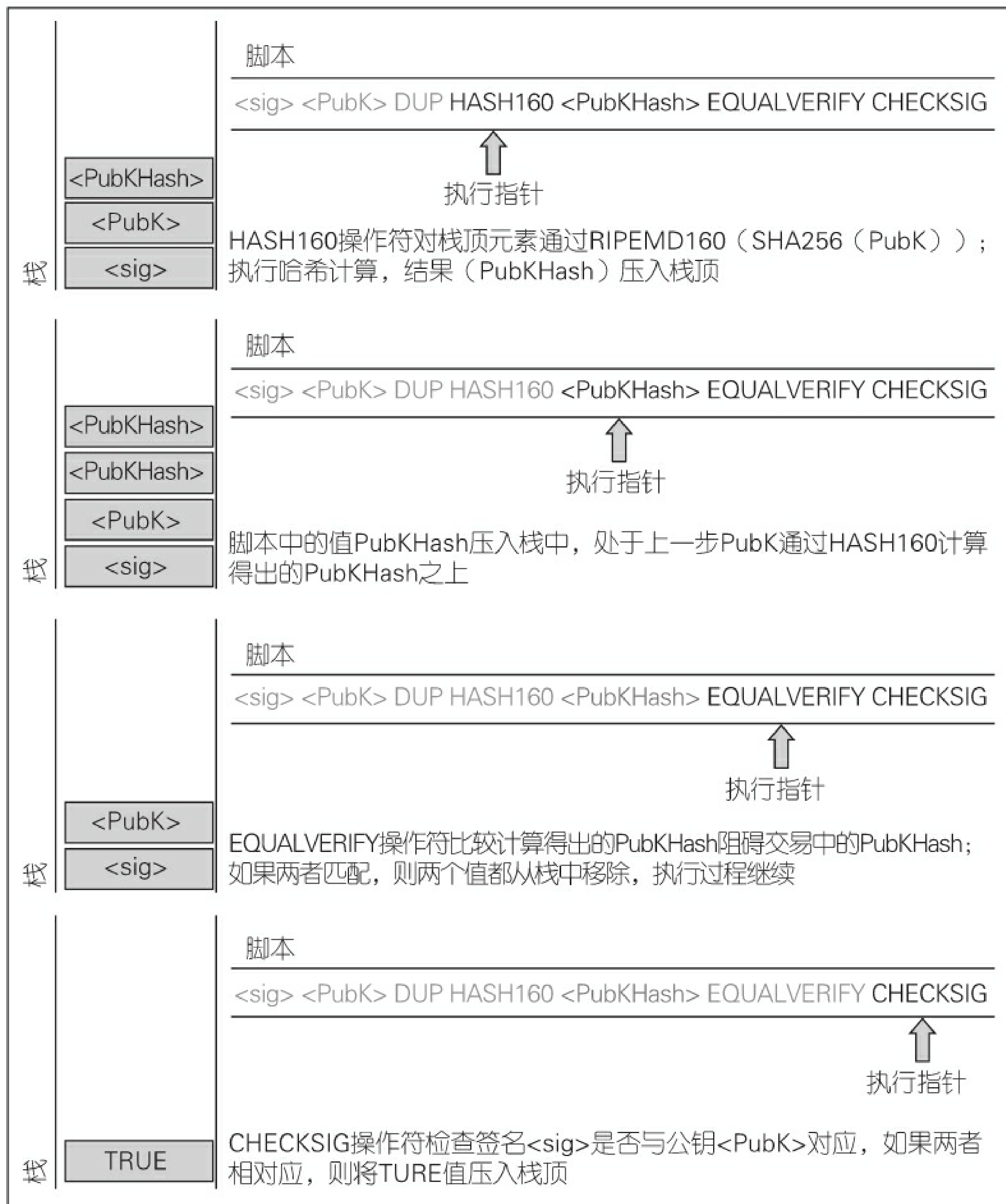


图5.4　评估一个P2PKH交易的脚本（第2部分）

多重签名

多重签名脚本设定一个条件，使N个公钥被记录在脚本中，约定N个公钥中的至少M个提供签名才能解除阻碍。这种交易也被称为M-of-N方案，这里的N代表密钥的总数，而M是用于验证的签名的最少数目。举例来说，一个2-of-3的多重签名中，列表中的3个公钥代表3个潜在的签名人，他们中至少要有2人提供签名才能验证交易的有效性，并允许使用资金。在撰写本书时，标准多重签名脚本限定为最多允许列出15个公钥，也就是说，用户可以选择1-of-1到15-of-15间的多重签名或者这个范围内的任意组合。15个公钥的限制条件可能在本书出版后已经更新了，所以请检查isStandard（）函数，看一下目前可被网络接受的限制。

设置了M-of-N多重签名条件的锁定脚本的一般形式如下。

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG
```

其中，N是全部列出的公钥数量，M是用以解锁输出的最小签名数量。

一个设置了2-of-3多重签名条件的锁定脚本看起来就像下面这样。

```
2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG
```

以下包含2个签名的解锁脚本可以满足上述锁定脚本的条件。

```
OP_0 <Signature B> <Signature C>
```

同样地，如果提供的是3个公钥中另外2个公钥对应的私钥签名，也可以满足锁定条件。

由于最初的CHECKMULTISIG实现中存在一个漏洞（bug），会导致过多项目被推出栈顶，因此前缀OP_0是必须的，但是它仅充当占位符的作用，会被CHECKMULTISIG忽略。

两个脚本可以组合成一个验证脚本。

```
OP_0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C>
3 OP_CHECKMULTISIG
```

当且仅当解锁脚本符合锁定脚本设置的条件时，这个组合脚本的执行结果才为真。在本例中，设置的条件就是，解锁脚本能否提供与设置为阻碍的3个公钥中的任意2个相一致的私钥有效签名。

数据输出（OP_RETURN）

比特币的分布式和时间标记的账簿——区块链，拥有远超支付功能的应用潜力。很多开发者已开始尝试利用比特币系统安全和弹性的优势，并基于交易脚本语言来开发各种应用，比如数字公证服务、股权证明、智能合约等。早期对比特币脚本语言的应用开发，主要包括创建能够在区块链上记录数据的交易输出。比如，将文件的数字指纹记录到区块链上，使任何人都能通过引用交易，在特定日期建立文件的存在性证明。

使用比特币的区块链来记录与比特币支付不相干的数据，是一个有争议的话题。很多开发者认为这是滥用，希望能阻止这种应用的开发。其他一些人则认为它正是区块链强大能力的体现，希望鼓励这类实验行为。那些反对将与交易不相关数据记录到区块链的人认为，这种行为导致了“区块链臃肿”，加重了完全节点用户的负担，用户必须承受更高的磁盘存储费用，而这些额外数据本来并不是系统设计时有意规划的。此外，这种交易把目标地址当作一个20字节长度的自由格式字段使用，其创建的UTXO无法使用。由于地址被当作数据项使用，不与任何私钥对应，结果是UTXO永远无法使用，成为虚假的支付交易。这种做法导致内存中的UTXO集合不断变大，因为这些无法使用的交易永远也不能移除，比特币节点不得不把它们一直存放在内存中，成本更是远高于磁盘存储。

比特币核心的0.9版中，达成了一项共识，引进OP_RETURN操作符，OP_RETURN允许开发者添加40字节与支付无关的数据到交易输出

中。不像“虚假”的UTXO，OP_RETURN操作符创建一个显式的证明不可用的输出，它并不需要存储在UTXO集合中。OP_RETURN输出记录在区块链上，消耗磁盘空间，使区块链变大；但它们不存储在UTXO集合中，不会导致UTXO内存膨胀，从而减轻完全节点的内存成本负担。

OP_RETURN脚本的样式如下。

```
OP_RETURN <data>
```

数据部分限定为40字节，通常代表一个哈希值，比如SHA256算法的输出（32字节）。很多应用程序会在数据前加一个前缀来标识应用。比如，存在证明（**http://proofofexistence.com**）数字公证服务使用8字节的前缀“DOCPROOF”，这是个ASCII码的字符串，其十六进制的表现形式为：44f4350524f4f46。

必须记住，没有“解锁脚本”与OP_RETURN对应，也就是说，一个OP_RETURN输出是无法花费的。总之，你无法花费锁定在输出中的钱，所以也就没必要将其作为潜在可用的输出存在UTXO集合中，因为OP_RETURN已经是证明不可用的。OP_RETURN通常是一个零比特币金额的输出，因为任何赋予这样一个输出的资金都会永久丢失。脚本验证软件碰到OP_RETURN操作符会立即停止验证脚本的执行，并将交易设为无效。所以，如果你的交易输入中不小心引用了一个OP_RETURN输出，交易就是无效的。一个标准交易［通过isStandard（）检查的交易］只能使用一个OP_RETURN输出。但是，在一个交易中，OP_RETURN输出可以与任何其他类型的输出进行组合。

支付到脚本哈希（P2SH）

支付到脚本哈希（P2SH）于2012年被引入，是一个强大的新型交易，它极大地简化了复杂的交易脚本。为了解释P2SH的必要性，我们

来看一个实际的例子。

在第1章中，我们介绍了穆罕默德，一个迪拜的电子产品进口商。穆罕默德的公司账户广泛使用了比特币的多重签名脚本特性。多重签名脚本是比特币高级脚本功能中最常见的应用之一，是一个非常强大的特性。针对所有客户的支付，即“应收账款”，或被称为AR，穆罕默德的公司要求使用多重签名脚本进行支付。在多重签名的方案下，所有客户的支付款，至少要提供2个签名才能解锁，一个来自穆罕默德，另外一个来自他的合作伙伴或拥有备份密钥的代理人。多重签名方案为公司治理提供了管控手段，可以有效防止公司的资产遭遇盗窃、侵占或丢失。

以下是最终的脚本，相当长。

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3
Public Key> <Attorney Public Key> 5 OP_CHECKMULTISIG
```

虽然多重签名脚本功能相当强大，但使用起来也很笨重。因为要使用这个脚本，穆罕默德不得不在客户付款前与他们一一沟通告之这个脚本。每个客户也不得不使用特殊的比特币钱包以生成交易脚本，客户还需要了解如何利用这个脚本来生成交易。此外，最终生成的交易比简单交易要大5倍，因为这个脚本包含了非常长的公钥。超大交易的负担，将以交易费用的形式转嫁到客户头上。最后，这种大交易脚本会被保存到完全节点内存的UTXO集合中，直到它被花费掉。所有这些问题使复杂输出脚本在实践中难以推广。

开发支付到脚本哈希就是要解决这些实际困难的，使得复杂脚本的使用跟支付到比特币地址一样简单。对于P2SH支付，数字指纹（加密哈希）代替了复杂的锁定脚本。当一笔交易准备花费一个UTXO时，除了解锁脚本，它还要提供与哈希匹配的脚本。简而言之，P2SH的意思就是“支付到一个匹配这个哈希的脚本，此脚本将在花费这个输出时提供”。

在P2SH交易中，被哈希替代的锁定脚本也被称为赎回脚本，它与

锁定脚本不同，是在赎回时才提供给系统。表5.4显示了不带P2SH的脚本，表5.5显示的是具有同样功能的P2SH脚本。

表5.4　不带P2SH的复杂脚本

锁定脚本	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
解锁脚本	Sig1 Sig2

表5.5　带P2SH的复杂脚本

赎回脚本	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
锁定脚本	OP_HASH160 <20 – byte hash of redeem script >OP_EQUAL
解锁脚本	Sig1 Sig2 redeem script

从表5.4和表5.5可以看出，使用P2SH后，描述详细解锁条件的脚本（赎回脚本）并没有在锁定脚本中给出。相反，在锁定脚本中，只出现了哈希值，而赎回脚本则在稍后交易输出被花费时，才作为解锁脚本的一部分出现。由此，额外交易费的负担和交易的复杂度从发送者转移到了接收者（花费者）。

我们回过头看一下穆罕默德公司的例子，复杂的多重签名脚本及最终的P2SH脚本。

首先是用于接收客户支付款的多重签名脚本。

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3
Public Key> <Attorney Public Key> 5 OP_CHECKMULTISIG
```

如果将占位符替换为真实的公钥（以04开头的520位数字），你可以看到脚本已经变得非常长。

```
2
04C16B8698A9ABF84250A7C3EA7EE-
DEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F
569FD4193AF5DE762C58704A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1
FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49047E63248
B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0
279CFC10F1E8E8F3020DECDBC3C0DD389D99779650421D65CBD7149B255382ED7F78E946580657EE
```

```
6FDA162A187543A9D85BAAA93A4AB3A8F044DA-
DA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5043752580AFA1EC-
ED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA7
74D207D137AAB59E0B000EB7ED238F4D800 5 OP_CHECKMULTISIG
```

整个脚本可以被一个20字节长度的哈希值替代，其计算过程如下。首先使用SHA256哈希算法，然后对结果使用RIPEMD160算法。最终上述脚本的20字节哈希是。

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

一个P2SH交易使用以下脚本将交易输出锁定到哈希上，不再需要很长的脚本。

```
OP_HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e OP_EQUAL
```

就像你所看到的，这个新脚本比原来的短多了。“支付到这5个多重签名脚本”与P2SH交易的“支付到拥有这个哈希值的脚本”是同等的交易。客户向穆罕默德公司发起支付时，只需要将这个短得多的锁定脚本包含到他的支付中。当穆罕默德需要花费这笔UTXO时，只要提供原始的赎回脚本（其哈希值锁定了UTXO）和所需的解锁签名即可，如下所示。

```
<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG>
```

两个脚本经由两个阶段进行组合。首先，赎回脚本与锁定脚本比对，确保哈希匹配。

```
<2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG> OP_HASH160 <redeem scriptHash>
OP_EQUAL
```

如果赎回脚本的哈希相匹配，将会执行解锁脚本以解锁赎回脚本。

```
<Sig1> <Sig2> 2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG
```

支付到脚本哈希地址

P2SH特性的另外一个重要组成部分是将脚本哈希编码成一个地址的能力，该能力在BIP0013中得到定义。P2SH地址是Base58Check编码的20字节脚本哈希，就如比特币地址是公钥的20字节哈希。P2SH地址

使用版本前缀“5”，其Base58Check编码的地址以“3”开头。举例来说，穆罕默德的复杂脚本，经哈希计算并进行Base58Check编码后形成的P2SH地址变成了39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw。现在，穆罕默德可以将这个“地址”发给他的客户，客户可以使用几乎所有的比特币钱包软件来生成一个简单的支付，就像支付给比特币地址。前缀3只是告诉他们这个地址是一种特殊类型的地址，与之对应的是一个脚本而不是公钥。不管怎么说，它工作起来与支付到比特币地址是完全一样的。

P2SH地址隐藏了所有的复杂性，发起这笔支付的人不需要看到脚本。

支付到脚本哈希的优势

支付到脚本哈希的特性，相对直接在锁定输出时使用复杂脚本具有以下优势。

●交易输出中的复杂脚本被更短的数字指纹替代，使得交易规模更小。

●脚本可以被编码为一个地址，交易发送者及其钱包软件不再需要复杂的工作去实现P2SH。

●P2SH将构建脚本的负担从发起者转移给了接收者。

●P2SH将存储长脚本的负担从输出（在UTXO集合中，从而影响内存）转移到了输入（只存储在区块链上）。

●P2SH将长脚本数据存储的负担从当前（支付）转移到了未来（输出被花费时）。

●P2SH将长脚本交易费用的负担从发送者转移给了接收者，接收者

在使用资金时必须包含赎回脚本。

赎回脚本与**isStandard**验证

在比特币核心客户端0.9.2版本前，支付到脚本哈希被isStandard（）函数限定为标准比特币交易脚本类型。也就是说，在花费交易输出时，提供的赎回脚本只能是标准类型的一种：P2PK、P2PKH，或者多重签名，不包含OP_RETURN和P2SH自身。

0.9.2版本后，P2SH交易可以包含任何有效脚本，使得P2SH标准更具弹性，允许试验更多新奇、复杂的交易类型。

需要注意的是，因为赎回脚本直到试图花费P2SH输出时才会向网络提供，如果使用一个无效脚本的哈希锁定输出，它也一样可以被执行。但结果是你将无法使用交易输出，因为包含赎回脚本的花费交易会因含有无效脚本而无法被网络接受。这就带来了一个风险，即你可以把比特币锁定到一个无法使用的P2SH交易中。由于脚本哈希不会提供任何其所代表的脚本的提示，网络可以接受与无效赎回脚本关联的P2SH锁定。

P2SH锁定脚本包含一个赎回脚本的哈希，它不含任何赎回脚本内容的信息。P2SH交易将被认为有效并被接受，即使赎回脚本是无效的。你可能会以这种方式不小心锁定比特币，导致将来无法使用。

第6章 比特币网络

点对点网络架构

比特币是一种基于互联网的点对点网络架构。所谓点对点（P2P）是指加入网络中的所有计算机均互为对等关系。节点与节点是平等的，没有“特殊”性，所有节点共同承担提供网络服务的责任。网络节点间以一种“扁平”的网状拓扑结构互联。网络中没有服务器，没有中心化服务，没有层次化。处于点对点网络中的节点同时提供和消费服务，互惠互利。P2P网络具有天然的弹性、去中心化和开放的特点。一个P2P网络架构的典型例子就是早期的互联网本身，那时基于IP（网络之间互连的协议）网络之上的节点都是平等的。如今的互联网架构变得更具层次化了，但是网络互联协议仍然保持了扁平拓扑的本质。除了比特币，应用范围最广、最成功的P2P技术的应用应该就是文件分享了，包括作为先驱的Napster，以及作为最新架构演化的BitTorrent。

比特币的P2P网络架构不仅仅是拓扑结构的选择。比特币是一个特意设计的点对点数字货币系统，网络架构既是其核心特性的反映，也是其特性的基础。去中心化控制是其核心设计原则，只有通过扁平的、去中心化的P2P共识网络，才能实现和维护这套机制。

“比特币网络”是指运行比特币P2P协议的所有节点的集合。除了比特币P2P协议，还有其他协议存在，比如Stratum协议，用于挖矿以及轻量级或移动钱包。这些额外的协议由网关路由服务器提供，这些服务器自身使用比特币P2P协议接入比特币网络，通过提供网关功能，将网络扩展到那些运行其他协议的节点。比如，Stratum服务器通过Stratum协议，将Stratum挖矿节点与主比特币网络相连，将Stratum协议桥接至比特币P2P协议上。我们使用“扩展比特币网络”指代包含比特币P2P协议、矿池挖矿协议、Stratum协议，以及其他用于连接比特币系统组件的协议的整个网络。

节点类型与角色

虽然在比特币网络中的节点是平等的，但基于它们所支持的功能，它们可能充当了不同的角色。一个比特币节点是一系列功能的集合，包括：钱包、矿工、区块链数据库、路由节点。一个完全节点拥有全部4个功能，如图6.1所示。

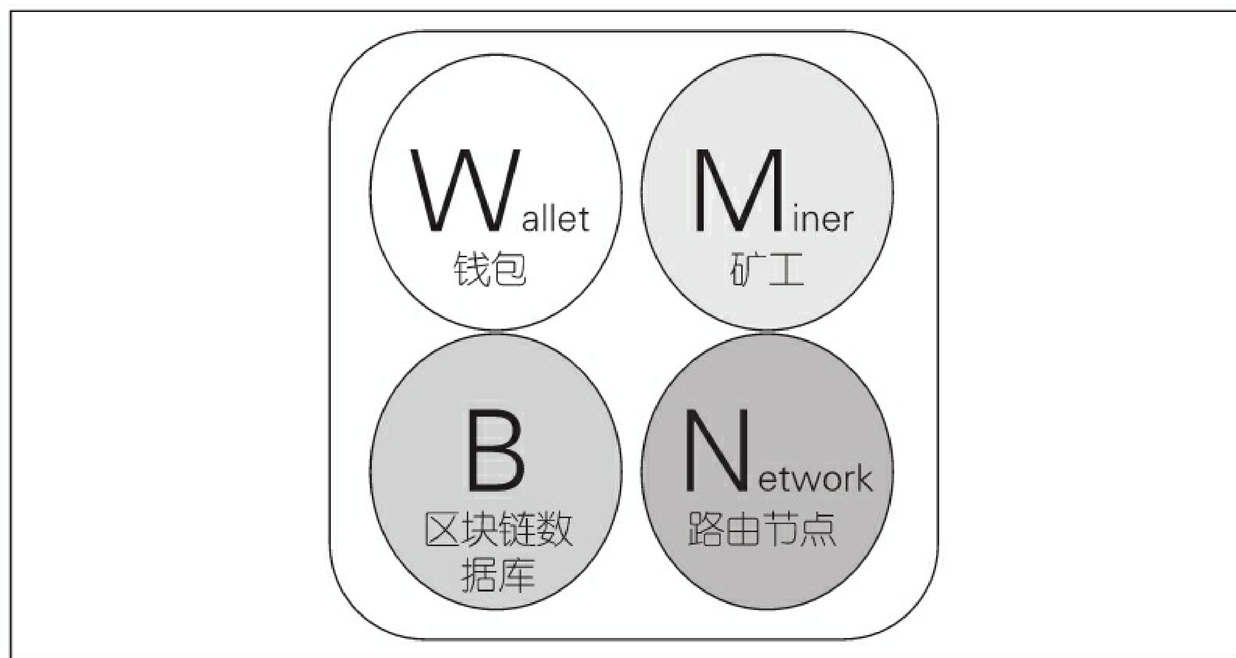


图6.1　比特币网络节点，拥有所有4项功能

所有节点均带有路由功能，从而能够加入网络，当然也可能包含其他功能。所有节点验证并传播交易和区块，发现并维护与其他节点的连接。在完全节点的例子中（见图6.1），路由功能被称为“网络路由节点”。

有些节点被称为完全节点，它们维护着一份完整的最新区块链副本。完全节点可以不依赖外部而自主权威地验证任何交易。而另一些节点只维护区块链的一个子集，它们验证交易时需要用到一种叫作简化支

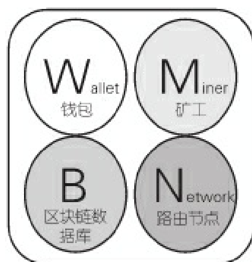
付验证（**Simplified Payment Verification**，简称**SPV**）的方法，这些节点也被称为SPV或轻量级节点。在完全节点的示例图中，完全节点的区块链数据库功能被称为“完全区块链”。在图6.3中，SPV节点没有区块链的全量副本。

挖矿节点采用特殊的硬件来求解工作量证明算法，它们通过竞争的方式创建新的区块。某些挖矿节点本身就是完全节点，维护一个完整的区块链副本，而另外一些则是轻量级节点，它们加入矿池，依赖矿池服务器来维护完全节点功能。在完全节点中，挖矿功能被称为“矿工”。

用户钱包可以是完全节点的一部分，这在桌面比特币客户端中比较常见。越来越多的用户钱包，特别是在类似智能手机等资源有限的设备上运行的钱包软件，则是SPV节点。钱包功能在图6.1中标注为“钱包”。

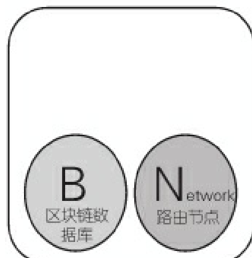
除了运行比特币P2P协议的主要节点类型，网络上还有一些服务器和节点运行其他协议，比如专业矿池协议、轻量级客户端访问协议等。

图6.2显示了在扩展比特币网络上最常见的几种协议类型。



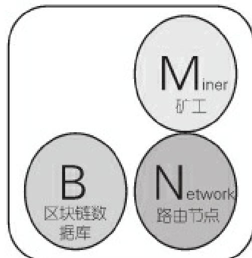
标准客户端（比特币核心）

包含钱包、矿工、区块链数据库，以及P2P网络上的路由节点。



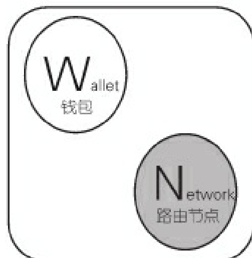
完全区块链节点

包含一个区块链数据库，一个P2P网络上的网络路由节点。



个体矿工

包含区块链数据库及比特币P2P网络路由节点，并提供挖矿功能。



轻量级（SPV）钱包

包含一个钱包、一个P2P网络上的网络节点，不含区块链。



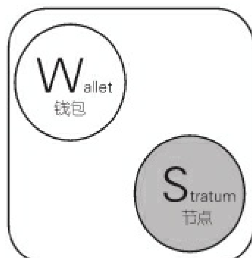
矿池协议服务器

网关路由器，将运行其他协议的矿池节点或Stratum节点与P2P网络相连。



矿工节点

包含基于Stratum协议或其他矿池挖矿协议的挖矿功能，不含区块链。



轻量级（SPV）Stratum钱包

包含钱包及基于Stratum协议的网络节点，不含区块链。

图6.2　在扩展比特币网络中不同类型的节点

扩展比特币网络

主比特币网络运行的是比特币P2P协议，大概包含7000到10000个运行不同版本比特币标准客户端（比特币核心）的节点，以及几百个运行着其他兼容比特币P2P协议软件的节点，这些软件包括BitcoinJ、Libbitcoin、btcd。在这些节点当中，只有少量节点同时也是挖矿节点，它们竞争挖矿，验证交易，并创建新的区块。各类大公司通过运行基于比特币核心的完全节点与比特币网络相连，它们拥有完整的区块链复制和网络节点功能，但是不具有挖矿和钱包功能。这些节点充当网络的边缘路由器，允许在顶层上构建各种其他服务，比如交易所、钱包、区块浏览器、商户支付处理等。

扩展比特币网络不仅包含前面介绍的运行比特币P2P协议的网络，也包含运行其他专门协议的节点。与比特币主网相连的矿池服务器和协议网关，将运行其他协议的节点连接到网络中。这些运行其他协议的节点主要是矿池节点（参看第8章）及轻量钱包客户端，这些节点均不保存全量区块链副本。

图6.3描述了含有各类型节点的扩展比特币网络，包括网关服务器、边缘路由器、钱包客户端，以及用于它们彼此相连的协议。

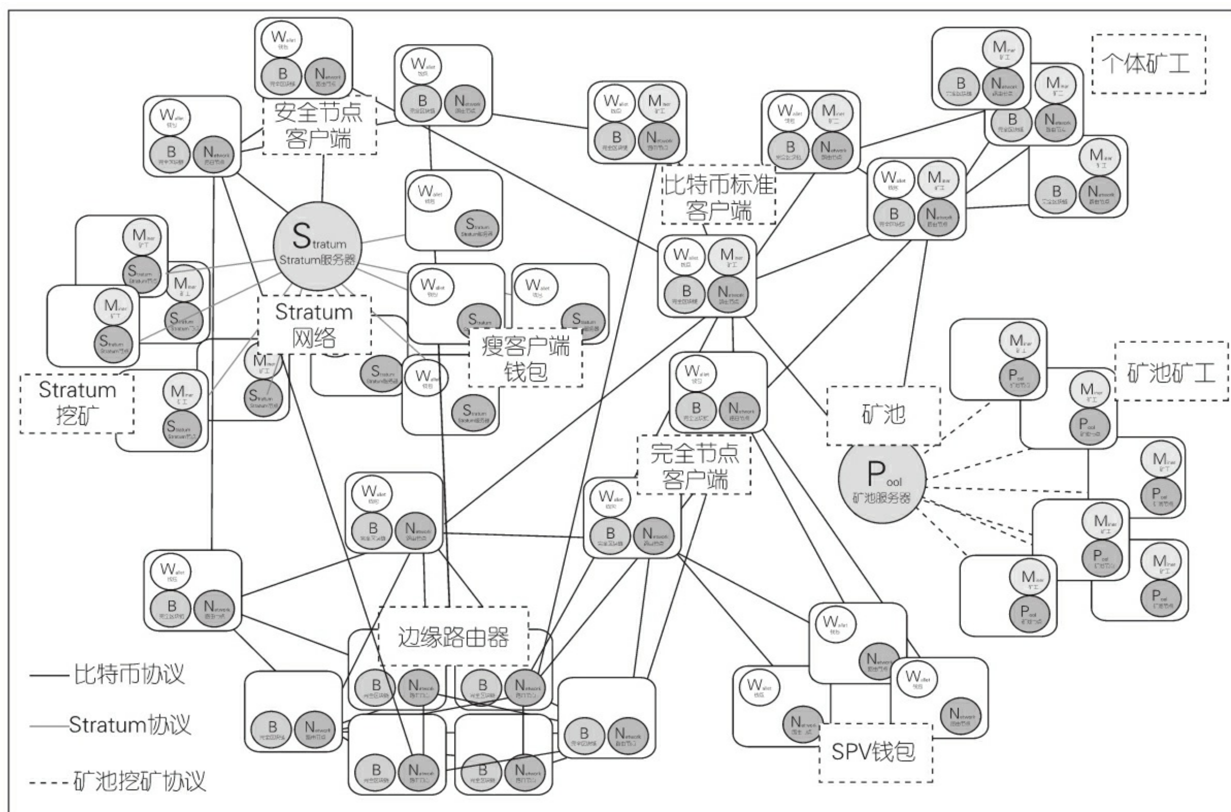


图6.3　描述了包括节点类型、网关和协议的扩展比特币网络

网络发现

当新节点启动时，首先需要找到网络中的其他比特币节点以加入进去。为开始这一流程，新节点必须在网络上找到至少一个节点，并与之连接。其他节点的地理位置无关紧要；比特币的网络拓扑不是以地理位置来定义的。因此，新节点可以随机选择任意存在的节点进行连接。

为了与已知节点相连，需要与之建立起TCP（传输控制协议）连接，比特币节点的服务端口通常为8333（约定俗成的比特币端口），或者其他约定的服务端口。一旦连接建立，新节点立即向服务端发送一个版本信息进行“握手（handshake）”。握手信息主要是识别信息，包括如下信息。

协议版本（**PROTOCOL_VERSION**）

定义比特币P2P协议版本的常量（比如：70002）。

本地服务（**nLocalServices**）

节点提供的本地服务列表，目前只有网络节点（NODE_NETWORK）。

时间（**nTime**）

当前时间。

对端地址（**addrYou**）

从本地节点看到的远端节点IP地址。

本地地址（**addrMe**）

本地节点发现的本机IP地址。

子版本（**subver**）

显示本地节点软件类型的子版本号（比如：“/Satoshi:0.9.2.1/”）+。

最佳高度（**BestHeight**）

本地节点区块链的高度。

［GitHub（**http://bit.ly/1qlsC7w**）上可看到版本网络信息的例子］。

远端节点返回一个verack消息进行应答，并建立连接，如果希望回连新节点并互换信息，也可以将它自己的版本消息发给新节点。

一个新加入的节点如何找到对等节点呢？一种方式是利用一系列被称为“**DNS**种子”的、能够提供比特币节点IP地址的DNS服务器进行查询。一些DNS种子可以提供稳定的比特币节点的静态IP地址列表；另一些则是BIND（Berkeley Internet Name Demon）协议的定制化实现，它们会返回通过爬虫收集到的或长期运行的比特币节点列表的一个子集。比特币核心客户端包含5个不同的DNS种子的名称。DNS种子所有者及实现方式的多样化，为初始运行过程提供了更高层次的可靠性保证。在比特币核心客户端中，是否使用NDS种子是由选项开关-dnsseed控制的，若设为1，就使用DNS种子，这也是默认设置。

反之，一个对网络完全不了解的自举节点，需要提供至少一个比特币节点的IP地址，通过该节点的介绍，实现与更多节点的相连。命令行参数-seednode可用于连接一个节点，并仅让该节点充当介绍节点，这种用法被称为DNS种子。当初始节点完成介绍任务后，客户端就可以与该节点断开，接着使用新发现的节点作为对等节点。对等节点间的首次握手如图6.4所示。

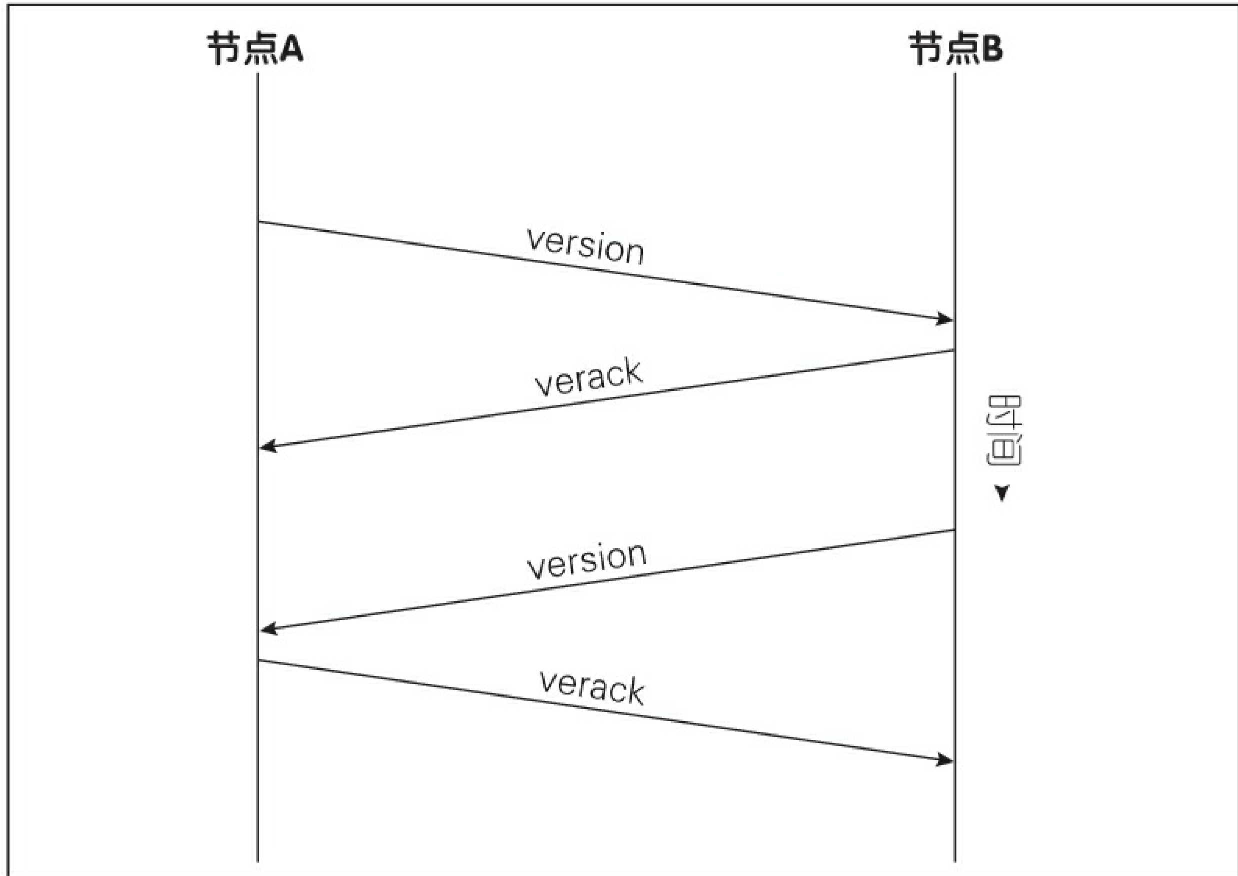


图6.4　对等节点间的首次握手

当一到多个连接建立起来后，新节点就向其所有邻居发送包含自身IP地址的addr消息。而邻居们则把该addr消息继续转发给自己的邻居，确保让更多节点知道新加入的节点，以便更好地连接。另外，新加入的节点也会发送getaddr给它的邻居，要求返回它们所知的对等节点的IP地址列表。通过这种方式，新节点就可以找到新的对等节点并与之连接，同时它也会在网络上传播它自己的信息，以便让其他节点找到。图6.5描述了地址发现协议。

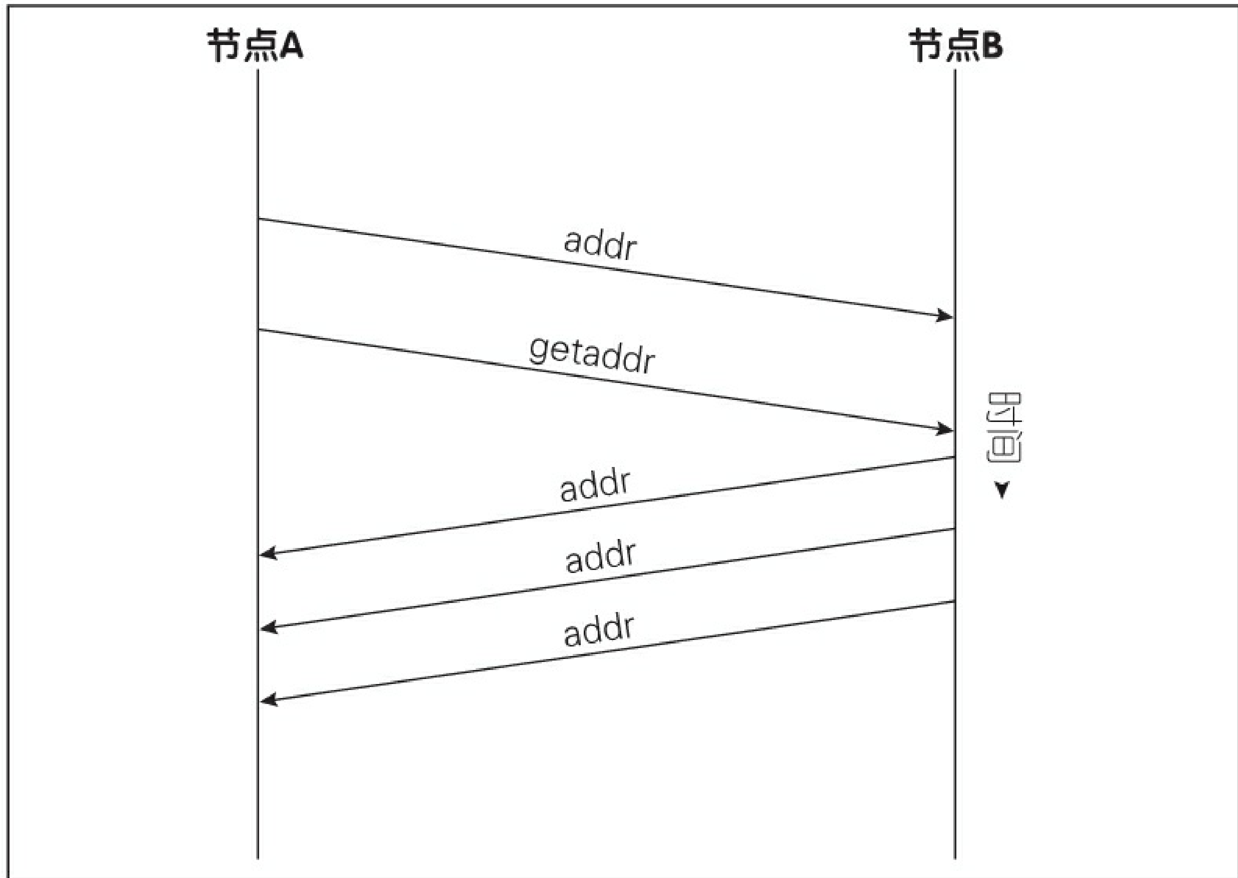


图6.5　地址的传播与发现

节点必须连接到不同的对等节点，以建立到达比特币网络的多条路径。路径总是不可靠的，节点会增加也会减少，所以节点必须不停地发现新的节点，以便在旧的连接丢失时可以建立新的连接，同时也需要帮助那些新加入的节点找到对等节点。启动时只需要一个连接即可，因为第一个对等节点会将其所知道的对等节点介绍给新加入的节点，与这些对等节点建立连接后，这些节点也会继续将它们所知道的对等节点信息告诉这个新节点。当然，连接太多的对等节点是没必要的，这是对网络资源的浪费。首次启动后，节点会记住它最近成功连接的节点，当节点重启时，它就可以快速重新建立与之前节点的连接。如果之前的任何节点都没有响应，节点可以利用种子节点重新连接。

在一个正在运行比特币核心的节点，利用命令getpeerinfo，可以列

出所有已知对等节点信息。

```
$ bitcoin-cli getpeerinfo

[
    {
        "addr" : "85.213.199.39:8333",
        "services" : "00000001",
        "lastsend" : 1405634126,
        "lastrecv" : 1405634127,
        "bytessent" : 23487651,
        "bytesrecv" : 138679099,
        "conntime" : 1405021768,
        "pingtime" : 0.00000000,
        "version" : 70002,
        "subver" : "/Satoshi:0.9.2.1/",
```

```
        "inbound" : false,
        "startingheight" : 310131,
        "banscore" : 0,
        "syncnode" : true
    },
    {
        "addr" : "58.23.244.20:8333",
        "services" : "00000001",
        "lastsend" : 1405634127,
        "lastrecv" : 1405634124,
        "bytessent" : 4460918,
        "bytesrecv" : 8903575,
        "conntime" : 1405559628,
        "pingtime" : 0.00000000,
        "version" : 70001,
        "subver" : "/Satoshi:0.8.6/",
        "inbound" : false,
        "startingheight" : 311074,
        "banscore" : 0,
        "syncnode" : false
    }
]
```

为覆盖自动节点管理，指定IP地址列表，用户可使用选项-connect=<IPAddress>指派一个或多个IP地址。一旦启用这一选项，节点将只与给定的IP地址相连，而不会自动发现并维护节点连接。

如果一个连接上没有网络流量，节点会定期发送一个消息以维持连接。如果节点在某个连接上超过90分钟没有通信，它将被认定为已与网络失去连接，需要寻找新的对等节点替代这个没有通信的节点。通过这种方式，网络会根据节点的变化和网络问题自动进行动态调整，从而达到不依赖中央控制而实现自主伸缩的目的。

完全节点

完全节点是指那些维护了包含所有交易的区块链全量副本的节点。更确切地说，它们应该被称为“完全区块链节点”。早期的比特币系统，所有节点都是完全节点；现在，比特币核心客户端依然是一个完全区块链节点。在过去的两年中，出现了新的比特币客户端，这些客户端不维护全量区块链，只是运行一个轻量级的客户端。我们将在下一节中更详细解释这种节点类型。

完全区块链节点维护一个包含所有交易的、完整的最新比特币区块链副本，它们可以独立地创建和验证区块链，从第一个区块（创世区块）一直创建到网络中最新的已知区块。完全区块链节点可以自主权威地验证任何交易，而不需要借助其他节点或其他信息来源。完全区块链节点依赖网络获取新交易区块的更新信息，验证并加入本地区块链副本中。

运行完全区块链节点会给你带来纯粹的比特币体验：不用依赖或信任任何其他系统，对所有交易进行独立验证。要判断你是否处在运行完全节点非常容易，因为它需要超过20G的永久存储（磁盘空间）来保存全量区块链。如果你需要很大的硬盘空间，并且需要2～3天才能与网络同步，那么你运行的是一个完全节点。这是摆脱中央集权、获得独立和自由的代价。

也有一些完全区块链比特币客户端存在替代实现，使用不同的编程语言和软件架构实现。但是，最常见的实现还是标准客户端即比特币核心，也被称为中本聪客户端。比特币网络中超过90%的节点运行着不同版本的比特币核心。在客户端发送的子版本字符串中，以及在getpeerinfo命令的结果显示中，它被标识为“Satoshi”，比

如“/Satoshi:0.8.6/”。

交换“库存”

完全节点与其他节点连接后，它要做的第一件事就是尝试创建一个完整的区块链。如果这是一个全新节点，本地完全没有区块链数据，那么它将只知道一个区块，即创世区块，它被静态内嵌到客户端软件里。从0号区块（创世区块）开始，新节点必须下载成千上万的区块来实现与网络的同步，并重建本地区块链。

同步区块链的过程始于版本（version）消息，版本消息中包含了最佳高度（BestHeight）信息，它是节点当前的区块链高度（区块数量）。节点从其对等节点获取version消息，了解它们有多少区块，并与自身区块数量进行比较。互联的对等节点首先交换一个getblocks消息，消息包含各自本地区块链最顶部区块的哈希（指纹）。如果某个节点发现接收到的哈希不等于区块链最顶部区块的哈希值，它就可以判断出接收到的哈希不属于最新区块，而是一个比较老的区块，从而判断自身的区块链比对等节点更长。

拥有更长区块链的节点，其区块数量比对等节点更多，因而能识别哪些区块是对端需要“追赶”的。它将识别出第一批需要分享的500个区块，通过inv（库存，inventory）消息将这500个区块的哈希传播出去。缺失这些区块的节点通过发送一系列getdata消息，并根据从inv消息获取到的哈希，请求完整的区块数据。

举个例子，我们假设一个节点只有创世区块。它从对等节点接收到一个inv消息，包含了后继500个区块的哈希值。接下来，这个节点将向所有相连的节点请求区块数据，为防止将单一节点压垮，它会把负载分散到不同节点上。节点跟踪每个对等节点正在“传输”的区块数量（已发送请求，尚未接收完成的），检查其是否超过限制

（MAX_BLOCKS_IN_TRANSIT_PER_PEER）。这样，如果一个节点需要获取大量的区块，它只会在早先的请求完成后才发送新的请求，如此一来，节点就能够控制更新节奏，避免压垮网络。当区块被接收后，它被加入区块链（我们将在第7章看到相关介绍）。随着本地区块链的逐步建立，更多的区块将被请求和接收，整个过程将一直持续到这个节点完成与全网络的同步。

不管与网络断开多长时间，一旦重新连接，节点都要重新进行本地区块链与对等节点的比较过程，从而获取丢失的区块。不管是离线几分钟、丢失几个区块，还是离线一个月、丢失几千个区块，节点都要从发送getblocks开始，获取inv应答，并下载丢失的区块（图6.6描述了库存和区块传播协议）。

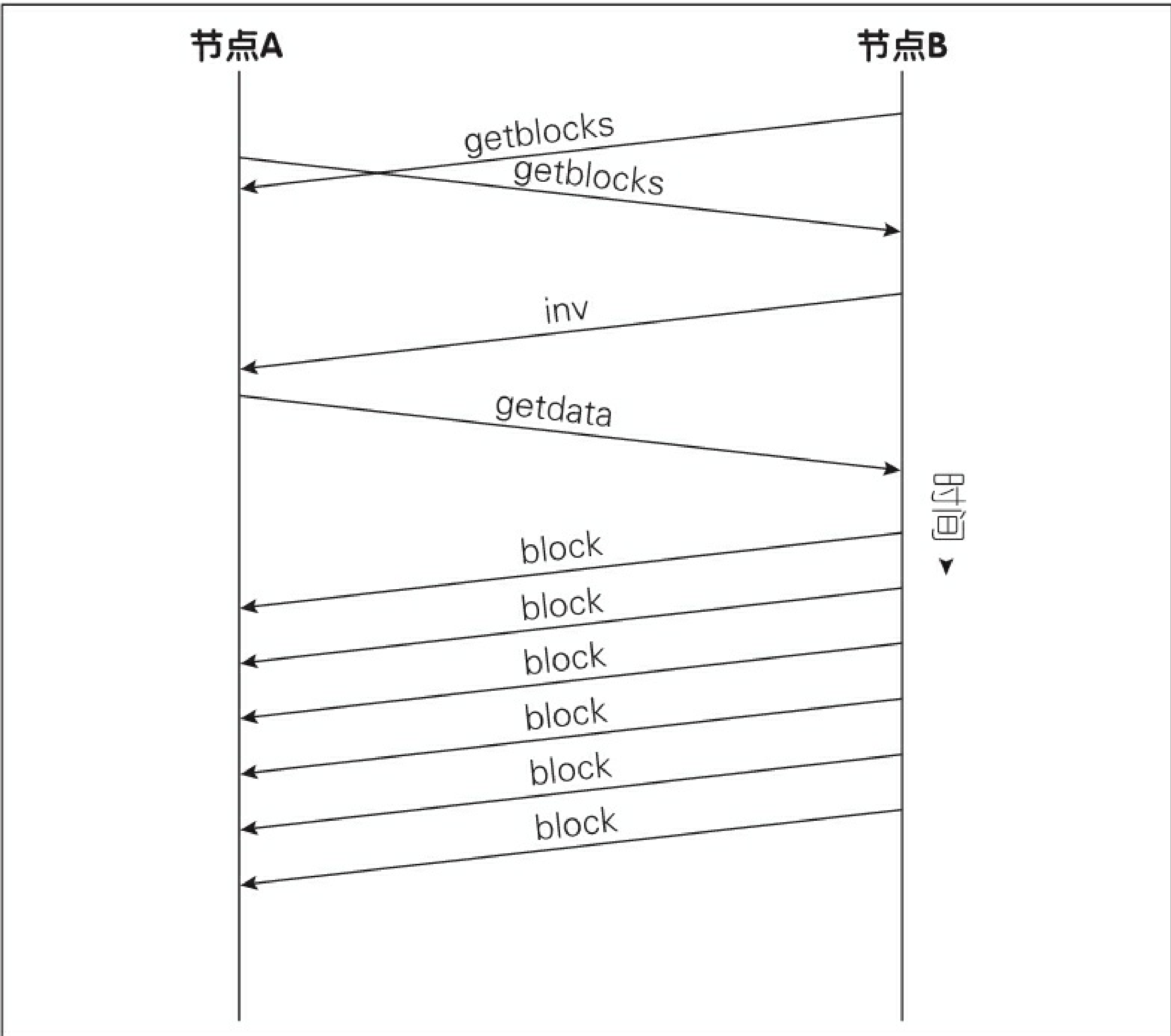


图6.6　节点通过从对等节点获取区块同步区块链

简化支付验证节点

不是所有节点都有能力存储完整区块链。很多比特币客户端是用来在空间、性能均有限的设备上运行的，比如在智能电话、平板电脑或嵌入式系统上运行。对于这些设备来说，SPV方法可以保证它们在不保存全量区块链的情况下也能正常运行。这些类型的客户端被称为SPV客户端或者轻量级客户端。随着比特币的应用越来越广泛，SPV节点已逐渐成为比特币节点的最常见形式，尤其是比特币钱包。

SPV节点只需要下载区块头，而不用下载每个区块中的交易。这种不带交易的区块链，其大小要比完全区块链小1000倍。SPV节点不能全景展示所有可花费UTXO的完整视图，因为它们并不了解网络上的所有交易。SPV节点使用一套稍有不同的方法验证交易，这种方法依赖对等节点按需提供相关区块链的局部视图。

作为类比，完全节点就像一个处在陌生城市但带了一张包含所有街道、所有地址的详细地图的游客，而SPV节点就像另一个同在陌生城市的游客，但他只知道一条主干道，通过随机询问陌生人来进行路线规划。虽然两个游客都能通过实地考察验证街道是否存在，但没有地图的游客不知道每条小巷中有些什么，也不知道附近还有其他什么街道。站在教堂街23号前，没有地图的游客无法知道这个城市是否还有其他“教堂街23号”的地址，也不知道这个地方是不是就是自己要找的那个。对于没有地图的游客来说，最好的办法就是询问足够多的人，并希望不会遭到抢劫。

简化支付验证通过引用交易在区块链中的深度（**depth**）而不是它们的高度（**height**）来验证交易。完全区块链节点则创建一条完整的、验证过的区块链，这条链由区块和交易组成，并按时间倒序一直延伸到

创世区块。一个SPV节点会验证所有区块的链（但不是所有的交易），并且把链和有关感兴趣的交易进行关联。

比如，当检查区块300000中的某个交易时，完全节点将300000个区块连接在一起，直到创世区块，由此构建了一个完整的UTXO数据库，通过验证UTXO未被花费，来确定交易的有效性。SPV节点无法确定UTXO是否已被花费，不能直接判断UTXO的有效性，因此SPV节点采用的验证方法不同。首先，利用墨克尔路径（**merkle path**）（参见第7章中“默克尔树”）建立交易和包含这笔交易的区块间的关联关系。然后，SPV节点一直等到序号从300001到300006的6个区块堆叠在该交易所在的区块之上，并通过确定交易的深度是在第300006区块到第300001区块之下，来验证交易的有效性。事实上，网络上的其他节点接受了区块300000，并在其上创建了额外的6个区块，根据代理协议，就可以证明交易不是一个双重支付交易。

当交易不存在时，SPV节点不会误认为交易在区块中存在。它通过请求默克尔路径证明，验证区块链中的工作量证明来确保交易存在于区块中。但是，交易存在性可以对SPV节点进行“隐藏”。SPV节点可以明确证明交易的存在性，但无法验证一个交易（比如同一个UTXO的双重支付交易）是不存在的，因为这类节点没有保存全部交易的记录。这个弱点可被用于针对SPV节点的拒绝服务攻击或双重支付攻击。为了防范这类攻击，SPV节点需要随机连接几个节点，以确保至少与一个诚实节点保持联系。这种随机连接需求意味着，当SPV节点只到虚假节点或虚假网络的连接，而没有到诚实节点或真实比特币网络的连接时，它们仍然是网络分区攻击或女巫攻击的脆弱环节。

对于大多数实际应用来说，只要能确保与网络保持良好连接，SPV节点就是足够安全的，这很好地平衡了资源、实用性和安全性的需求。若需要保证绝对安全性，那就只能选用完全区块链节点，完全节点相比SPV节点要更加安全。

完全区块链节点通过检查区块链中交易所在区块以下的所有区块来验证交易，确保UTXO尚未被使用，SPV节点则通过计算交易所在区块之上的区块数量，来检查交易被埋了多深。

为了获取区块头，SPV节点使用getheaders请求消息来取代getblocks消息。收到请求的对等节点使用headers消息发送区块头，一次最多发送2000个区块头。这与完全节点获取完全区块的过程是一样的。SPV节点在与对等节点连接的链路上设置一个过滤器，过滤对等节点发送来的区块和交易数据流。对于感兴趣的交易，则使用getdata请求进行获取。对等节点生成一个包含交易的tx消息作为应答。图6.7展示了区块头的同步过程。

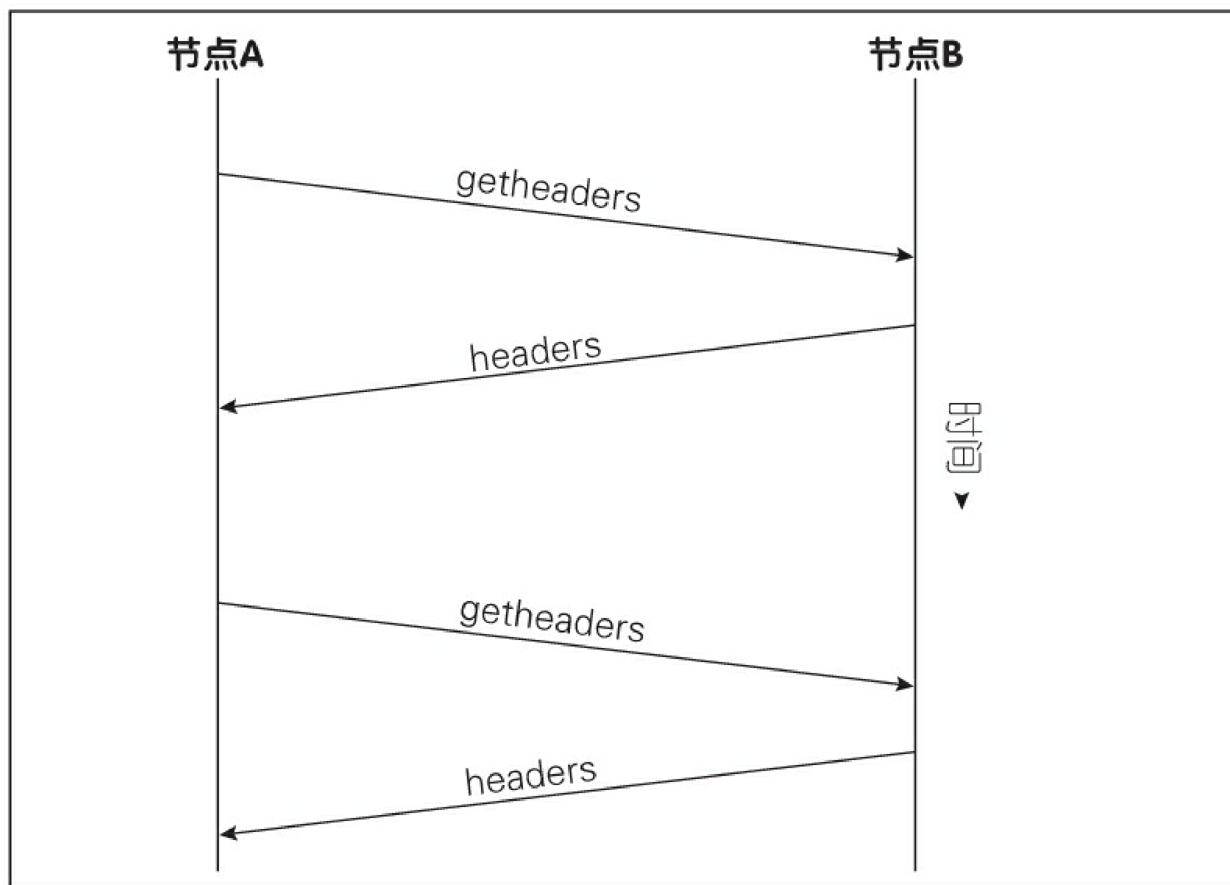


图6.7　SPV节点同步区块头

因为SPV节点需要获取特定的交易以有选择地验证交易，这就给隐

私带来了威胁。不像完全区块链节点收集每个区块中的所有交易，SPV节点对特定数据的请求，会无意中泄露它们钱包中的地址。比如，第三方通过持续监视网络，就可以跟踪一个SPV节点钱包发送的所有交易请求，从而将这些请求与用户钱包的比特币地址进行关联，达到侵犯用户隐私的目的。

SPV节点被引入不久，开发者又新增了一个叫作布隆过滤器（**bloom filters**）的功能，用以处理SPV节点的隐私问题。布隆过滤器通过概率而不是固定匹配模式的机制，使SPV节点可以接收交易子集而不用暴露确切的感兴趣地址。

布隆过滤器

布隆过滤器是一个概率搜索过滤器，采用一种不精确指定的方式来描述期望的匹配模式。布隆过滤器提供了一种在保护隐私的前提下表达搜索模式的有效途径。SPV节点使用这种方式向其对等节点请求匹配特定模式的交易列表，而不用暴露它们搜索的确切地址。

在我们之前类比的例子中，一个没有地图的游客询问到特定地点“教堂街23号（23 Church St.）”的路线。如果他向一个陌生人询问到该街道的路线，无意间就暴露了他的目的地。如果使用布隆过滤器，他可能问的就是“这附近是否有条街道，它的名字以R-C-H结尾？”这种提问方式，暴露的信息就要比直接说“教堂街23号”要少一些。通过这项技术，游客可以使用较详细的信息，如“以U-R-C-H结尾”来描述地址，也可以使用如“以H”结尾这样更简短的信息。通过改变搜索的精确度，游客可得到更多或更少的信息，相应的代价就是获取更精确或更模糊的结果。如果提供模糊的信息，他可以更好地保护隐私，但将得到非常多的地址，而大多数地址都是不相干的。如果提供相对精确的信息，得到的地址则较少，但在保护隐私上也会较弱。

布隆过滤器允许SPV节点通过调节搜索条件的精确度来提供这项服务。更明确的布隆过滤器将产生更精确的结果，但是代价是暴露用户钱包中使用的地址。粗略的布隆过滤器则因匹配更多的交易而带来更大的数据量，这些交易大多与本节点无关，但是可以为节点提供更好的隐私保护。

SPV节点初始化时把布隆过滤器设置为“空”，在此状态下，布隆过滤器不匹配任何模式。接着，SPV节点生成一个钱包中所有地址的列表，并创建一个匹配所有地址的交易输出的搜索条件。通常，每个搜索

条件就是“发送到公钥哈希”的脚本，这个脚本实际上就是出现在每个发送到公钥哈希（地址）的交易输出上的锁定脚本。如果SPV节点正在跟踪一个P2SH地址的余额，那么搜索条件就是“支付到脚本哈希”的脚本。接下来，SPV节点将这些条件添加到布隆过滤器中，使过滤器能够在符合搜索条件的情况下识别出交易。最后，把布隆过滤器发送给对等节点，对等节点依据设定条件将匹配的交易传到本地SPV节点。

布隆过滤器在实现过程中由一个包含N比特位（N位域）的可变长度数组和M个哈希函数构成。哈希函数设置成输出总是在1到N之间，与二进制数组长度对应。哈希函数是确定的，因此，任何实现布隆过滤器的节点都使用相同的哈希函数，并且在输入确定的情况下，将得到相同的结果。通过选择不同长度（N）的布隆过滤器，选用不同数量（M）的哈希函数，布隆过滤器可以调整精确程度和隐私保护级别。

在图6.8中，我们使用一个很小的16位数组，以及一个包含3个哈希函数的集合，演示布隆过滤器的工作过程。



图6.8　简单布隆过滤器的例子，使用16位域和3个哈希函数

布隆过滤器初始化时，二进制数组被设置为全零。为了向布隆过滤器中添加新的匹配模式，首先需要顺次利用预设的哈希函数对模式进行

计算。使用第一个哈希函数计算后，将得到一个1到N间的数字，然后将数组上对应的比特位（从1到N编号）设置为1，从而记录哈希函数的输出。接着，使用第二个哈希函数设置数组的第二个比特位，以此类推。一旦M个哈希函数都计算完成后，搜索模式将被“记录”在布隆过滤器上——即二进制数组中的M个比特位被从0改为1。

图6.9是向图6.8所示的简单布隆过滤器中添加一个模式“A”的例子。

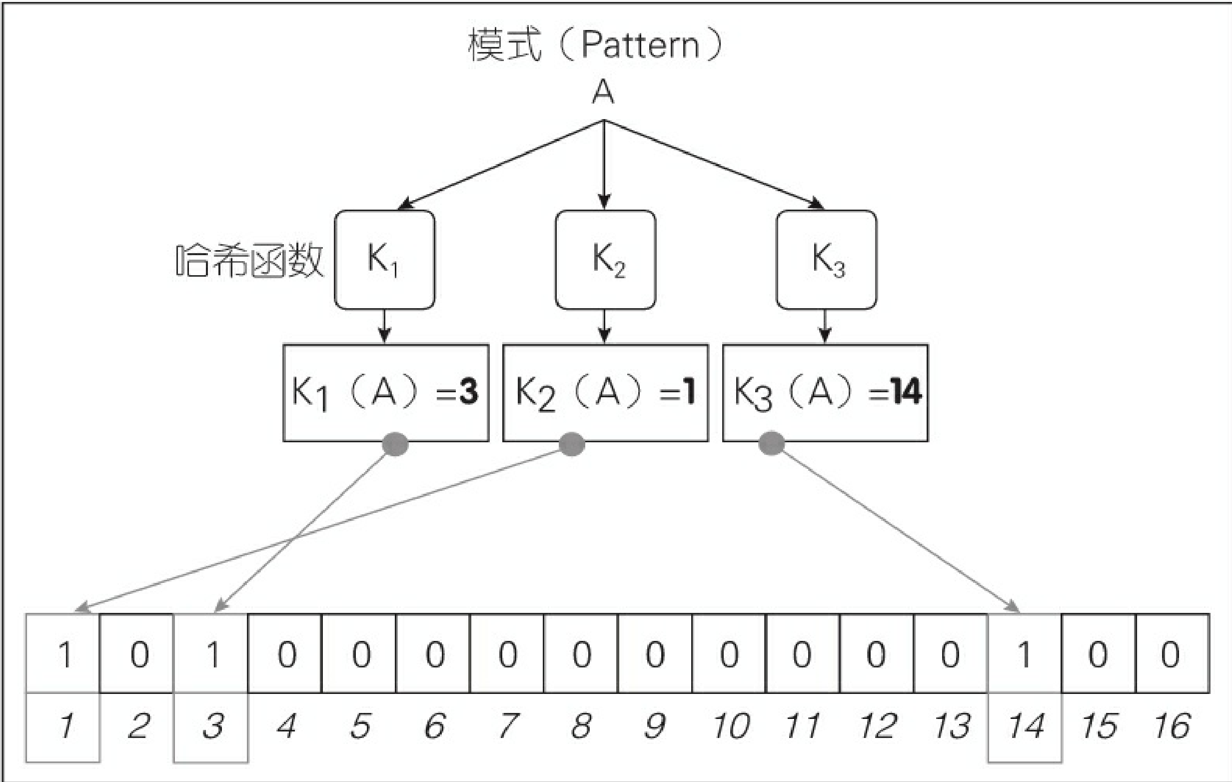


图6.9　添加模式“A”到简单布隆过滤器中

添加新的模式非常简单，只是重复一下刚才的步骤。模式被每个哈希函数顺序计算，然后在二进制数组相应位置设置1以记录哈希结果。注意，当采用更多的哈希函数时，可能出现多个哈希结果一样的情况，这时该比特位维持为1不变。实际上，随着模式的增多，越来越多的哈希结果会被记录在相同的位置上，过滤器也因设置为1的位置变多而开始变得饱和，准确性也相应降低了。这就是为什么布隆过滤器是一种概率数据结构的原因——它会随着更多的模式加入而变得不精确。精确度

依赖添加的模式数量、二进制数组大小（N）及哈希函数数量（M）三者关系。更大的二进制数组、更多的哈希函数可以在较高精确度的情况下记录更多的模式；而较小的二进制数组或者更少的哈希函数只能记录较少的模式，相应的精确度也较低。

图6.10是添加第二个模式“B”到简单布隆过滤器的例子。

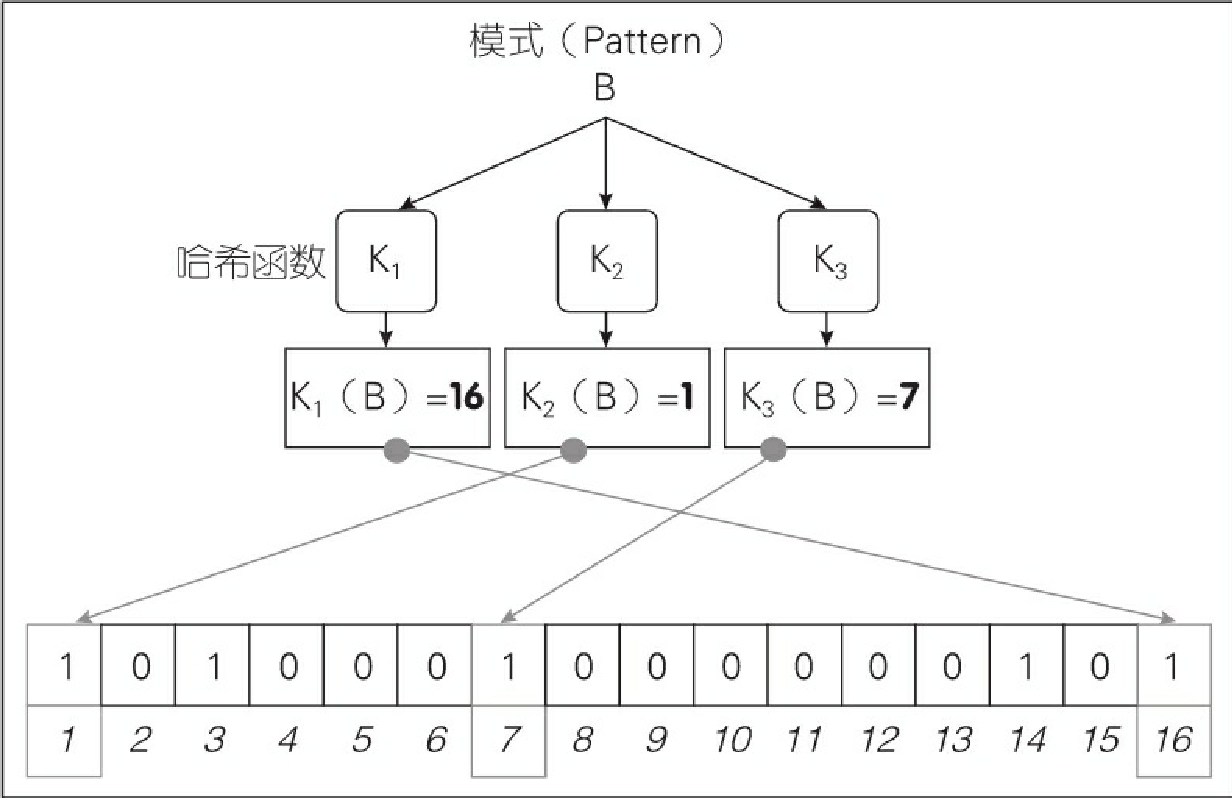


图6.10　添加模式“B”到简单布隆过滤器中

为测试一个模式是否为布隆过滤器的一部分，用M个哈希函数依次对模式进行计算，并用其结果与二进制数组进行比对。如果数组中所有索引号等于哈希结果的位均设为“1”，那么这个模式很可能已被记入布隆过滤器。因为这些位也可能是其他模式的哈希结果的重叠，答案是不确定的，但确实有可能性。简而言之，布隆过滤器的正匹配代表“可能是的”。

图6.11是测试模式“X”是否在简单布隆过滤器中存在的例子。其相

应的比特位均已设为“1”，因此模式可能是匹配的。

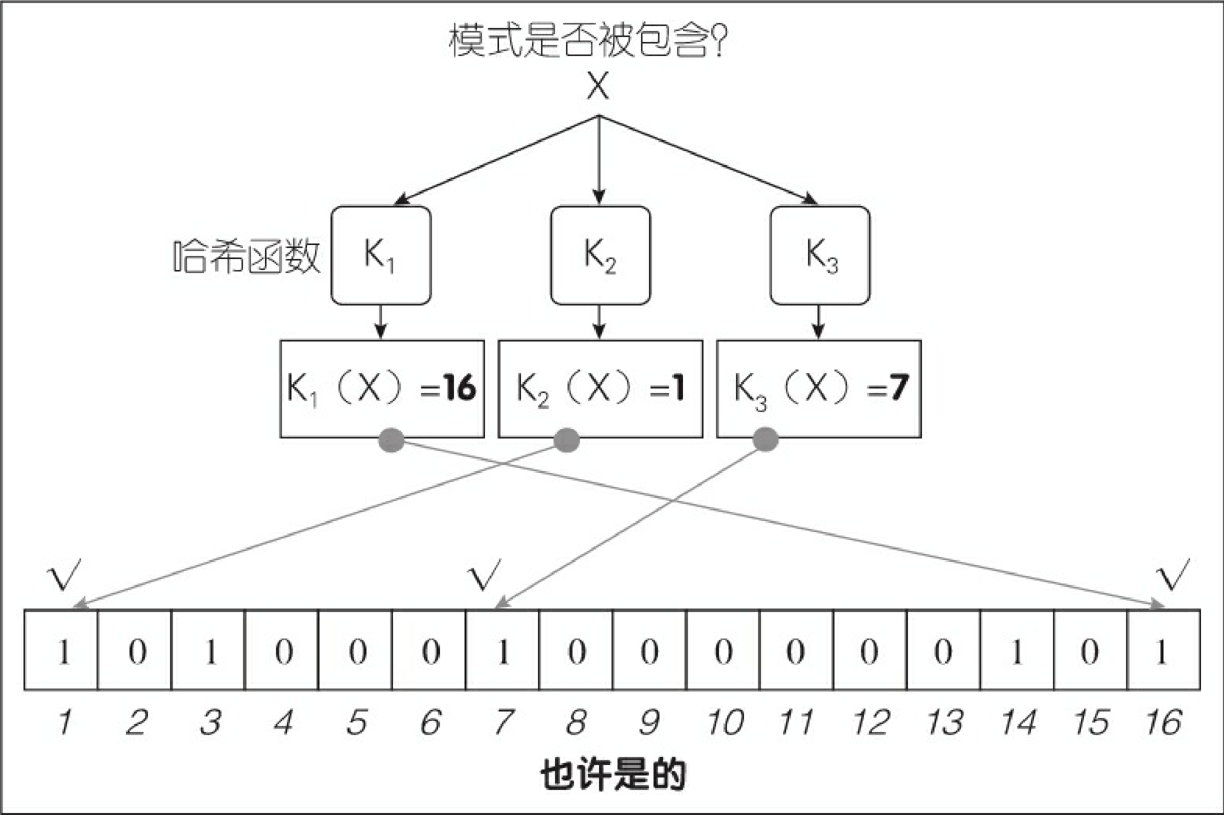


图6.11　测试模式“X”是否在布隆过滤器中，结果是一个概率正匹配，意思是“可能”

相反，如果模式与布隆过滤器测试过后，某些位被设置为0，则可以证明模式没有被布隆过滤器记录。否定的结果不是可能，而是确定。简单地说，布隆过滤器上的负匹配意味着“肯定不是！”

图6.12是测试简单布隆过滤器中是否存在模式“Y”的例子。其中有一位被设成了0，则此模式一定是不匹配的。

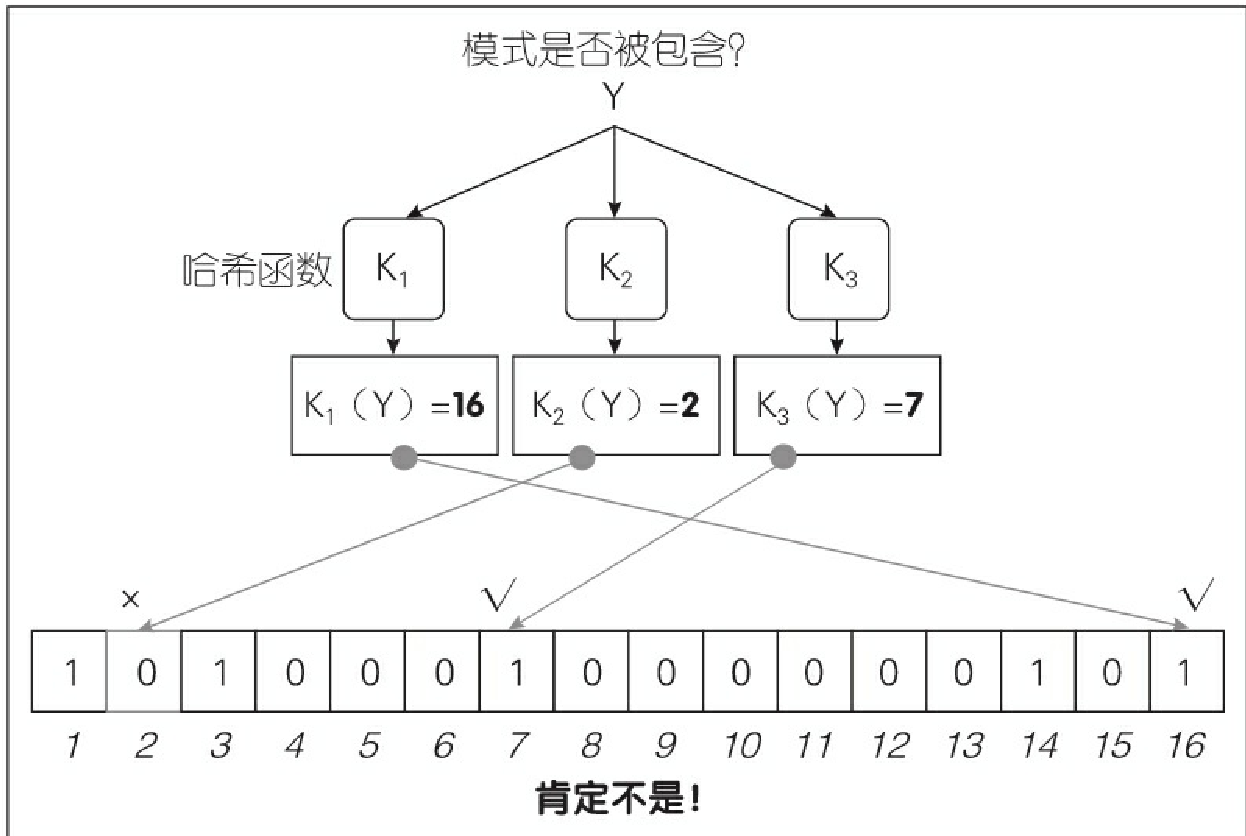


图6.12　测试模式“Y”是否在过滤器中，结果是确定不匹配，意味着“肯定不是！”

布隆过滤器在比特币中的实现方式，在比特币改进提案37（BIP0037）中有所描述。参见附录B或访问GitHub（http://bit.ly/1x6qCiO）。

布隆过滤器与库存更新

SPV节点使用布隆过滤器过滤从它的对等节点接收到的交易（以及包含它们的区块）。SPV节点首先创建一个用于匹配其钱包中所有地址的过滤器。然后，SPV节点发送一个过滤器加载（filterload）消息给其对等节点，消息中包含了需要连接时使用的布隆过滤器。当过滤器建立起来后，对等节点依据过滤器测试每个交易的输出。只有匹配过滤器的交易才发送给SPV节点。

作为对SPV节点getdata消息的应答，对等节点向其发送默克尔区块（merkleblock）和匹配交易的默克尔路径（参看第7章“默克尔树”），其中merkleblock消息仅包含那些匹配过滤器的区块头。对等节点同时也传送包含匹配过滤器的交易信息给SPV节点，使用tx消息格式。

设置过滤器的节点也可交互式地增加新的模式到过滤器中，通过发送过滤器增加（filteradd）消息实现。要清除过滤器，节点可以发送过滤器清除（filterclear）消息。由于无法从过滤器中移除单个模式，当不再需要某个模式时，需要通过清除过滤器并重新发送过滤器的方式进行更新。

交易池

比特币网络中几乎每个节点都会维护一个临时的未确认交易列表，被称为内存池（**memory pool**）或交易池（**transaction pool**）。节点使用这个池子对那些已发布到网络但尚未被包含进区块链的交易进行跟踪。比如，持有用户钱包的节点可以利用交易池，跟踪发送到用户钱包但尚未确认的支付交易。

随着交易被接收和验证，它们被加入交易池并且被中继到相邻节点，从而在网络中实现传播。

有些节点也维护一个独立的孤儿交易池。如果一个交易的输入引用了一个尚处未知状态的交易，比如说父交易缺失，这个孤儿交易就将被临时存储在孤儿交易池中，直到其父交易抵达本节点。

当交易被加到交易池时，将对孤儿交易池进行检查，以发现是否有交易引用了这个交易的输出（即其子交易）。若匹配，孤儿交易就会通过验证，并从孤儿交易池被移到普通交易池，交易链条将被补全。鉴于新移入的交易不再是孤儿交易，处理进程将递归寻找新的后代交易，直到再也找不到更多的后代。一个父交易进来后，通过重新组合孤儿交易与父交易的关系，会触发互相依赖的交易链的连锁重建。

不管是交易池还是孤儿交易池（如果实现了的话），都存放于内存中，而不是保存到永久存储中；它们随着接收到的网络消息而被动态填充。当一个节点启动时，两个池都是空的，随着不断从网络上接收新的交易，内存池也逐步被填上。

有些比特币客户端在实现过程中还维护一个UTXO数据库或UTXO池，这是一个区块链上的未花费交易输出的集合。虽然“UTXO池”听起

来跟交易池类似，但是它代表的是一个完全不同的数据集合。不像交易池或孤儿交易池，UTXO池不会被初始化为空集合，而是包含几百万条未花费输出，甚至包括一些日期回溯到2009年的交易。UTXO池可保存在本地内存，也可以存储在持久化的、带索引的数据库中。

然而，交易池或孤儿交易池都只是单个节点的本地视图，节点与节点间可能由于全新启动或者重启而产生巨大区别；UTXO池代表网络当前的共识，节点间的差异通常很小。另外，交易和孤儿交易池只包含未确认交易，而UTXO只包含已确认的交易输出。

警告消息

警告消息是一个极少使用的功能，但大多数节点均设置了该功能。警告消息是比特币的“紧急传播系统”。通过它，比特币核心开发者可以向所有的比特币节点发送紧急文本消息。当比特币网络发生严重问题时，这个特性使核心开发团队可以通知所有比特币用户，比如提醒用户注意新发现的严重bug。警告系统只被用过几次，影响最大的一次是在2013年早期，那时发生了一次严重的数据库bug，导致比特币区块链上出现一个多区块分叉。

警告消息通过alert消息传输。警告信息包含如下字段。

ID

标识警告信息，使得重复警告信息可被发现。

Expiration

警告到期时间。

RelayUntil

到期后警告消息不再被中继。

MinVer, MaxVer

警告消息适用的比特币协议版本范围。

subVer

警告消息适用的客户端软件版本。

Priority

警告级别，目前未启用。

警告消息使用公钥体系进行密码学签名。对应的私钥由选定的几个核心开发团队成员持有。数字签名可防止虚假警告消息通过网络传播。

每个接收到警告消息的节点都会对其进行验证，检查其有效期，然后继续向各自的邻居传播，确保消息可以很快地在全网中传播。除了传播警告消息，节点可能还会实现一些用户界面功能，将警告消息推送给用户。

在比特币核心客户端中，警告是通过命令行选项-alertnotify进行配置的，该选项允许用户指定收到警告后需要运行的命令。警告消息以参数的形式，将其传给alertnotify指定的命令。最常见的方式是将alertnotify的指定命令设置为生成一个电子邮件消息，发送到节点的管理员，电子邮件内容为警告消息。若图形用户界面客户端（bitcoin-Qt）正在运行，警告消息也将以弹出对话框的方式显示给用户。

其他一些比特币协议的应用，可能以不同的方式处理警告消息。很多嵌入式硬件挖矿系统则不支持警告消息功能，因为它们没有用户界面。强烈建议运行此类挖矿系统的矿工向矿池经营者订阅警告信息，或者运行一个轻量级的节点专用于接收警告消息。

第7章 区块链

介绍

区块链数据结构是一种有序的、后向连接的交易区块列表。区块链既可存储于扁平文件，也可存储于简单数据库。比特币核心使用谷歌的LevelDB数据库存储区块链元数据。区块是“后向”连接的，每个区块都有链接指向链条上的前序区块。区块链通常可想象为一个垂直堆栈，新的区块堆叠在其他区块的顶部，第一个区块是堆栈的基础。区块堆叠在其他区块之上的形象比喻导致了一些名词的引入，比如，“高度（height）”指本区块到第一个区块的距离，“顶部（top）”或“顶端（tip）”指最新加入的区块。

区块链中的每个区块，在其头部使用一个通过SHA256加密哈希算法生成的哈希值进行标识。每个区块头还包含一个“前序区块哈希”的字段，对前序区块（父区块）进行引用。换句话说，每个区块在区块头中均存有父区块的哈希。将每个区块连接到其父区块的哈希序列形成了一条可以一直追溯到第一个区块（创世区块）的链条。

虽然一个区块只能有一个父区块，但它却可以临时拥有多个子区块。每个子区块都指向相同的父区块，在“前序区块哈希”字段中拥有相同的父区块哈希。多子区块的现象是区块“分叉（fork）”时才发生的临时状态，其原因是不同的矿工几乎同时发现了不同的新区块（参见第8章中“区块链分叉”）。“分叉”最终都会得以解决，只有一个子区块会成为区块链的一部分。虽然一个区块可以有多个子区块，但每个区块只能有一个父区块，这是因为区块只有一个“前序区块哈希”字段，指向它唯一的父区块。

“前序区块哈希”字段位于区块头部，因此会影响当前区块的哈希。如果父区块的标识改变，子区块的标识也会随之变化。当一个父区块以

任意方式改变时，父区块的哈希值必然跟着变化。而父区块哈希的改变，又迫使子区块“前序区块哈希”指针跟着变化。这必将导致子区块哈希的变化，进一步导致了孙区块到子区块指针的变化，从而孙区块哈希也必须改变，以此类推。这种串联影响可以确保一旦一个区块有了多个后代，除非重新计算所有的后续区块，这个区块就没法修改。因为重新计算需要极大的计算量，长区块链的存在使区块链中较深的历史区块不可修改，这是保证区块链安全性的重要特性。

一种思考区块链的方式是将其想象为地质构造层或者冰川核心样本。表层可能会因季节变换而有所改变，甚至还没沉淀就已经被风吹走。但是一旦深入到地下几十厘米，地质层就变得更加稳定了。如果深入几百米的地底进行考察，你将会发现一个几百万年未曾受过干扰的历史概貌。在区块链中，最新的几个区块可能会因分叉而出现重计算的情况。最新的6个区块就如同地质结构的表层。但是一旦更加深入区块链，超过6个区块后，区块被改变的可能性将越来越小。当往前回溯100个区块后，区域块链就已经变得非常稳定，以致铸币交易（产生新比特币的交易）也可以花费了。回溯几千个区块（差不多一个月）后，区块链就成了稳定的历史，再也无法改变。

区块结构

区块是一种数据结构容器，用以汇聚交易并加入公共账本——区块链。区块由包含元数据的区块头，以及紧跟其后的长长的交易列表组成。区块头80字节长，而普通交易最少250字节，一般一个区块中包含的交易超过500个。一个完整的区块，包括所有交易，其长度超过区块头的1000倍。表7.1描述了区块的结构。

表7.1　区块结构

大小	字段	描述
4 字节	区块大小（Block Size）	区块按字节计算的大小，不含本字段
80 字节	区块头（Block Header）	几个字段构成了区块头
1 ~9 字节（varInt）	交易计数器（Transaction Counter）	紧跟的交易数量
可变长度	交易（Transactions）	区块中的交易记录

区块头

区块头包含3个区块元数据集合。第一个是到前序区块哈希的引用，在区块链中将本区块与前面的区块相连；第二个是元数据集，即难度（**difficulty**）、时间戳（**timestamp**）、随机数（**nonce**），它们与挖矿竞争相关，将在第8章详述；第三个元数据集是默克尔树（merkle tree）的根，一个高效概括区块中所有交易的数据结构。表7.2描述了区块头的结构。

表7.2　区块头结构

大小	字段	描述
4 字节	版本（Version）	跟踪软件/协议更新的版本号
32 字节	前序区块哈希（Previous Block Hash）	对链中前序（父）区块哈希值的引用
32 字节	默克尔根（Merkle Root）	本区块所有交易的默克尔根的哈希
4 字节	时间戳（Timestamp）	本区块大致的创建时间（Unix 时间戳）
4 字节	难度目标（Difficulty Target）	本区块工作量证明算法的难度目标
4 字节	随机数（Nonce）	用于工作量证明算法的一个计数器

其中，随机数、难度目标、时间戳用于挖矿过程，将在第8章详细介绍。

区块标识符：区块头哈希和区块高度

区块的主标识符是它的加密哈希，或者称为数字指纹，通过对区块头运行两次SHA256计算得到。结果的32字节哈希值叫作区块哈希（**block hash**），但是称其为区块头哈希（**block header hash**）更准确，因为只有区块头被用于哈希计算。举例来说，000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f是第一个比特币区块的区块头哈希。区块哈希只唯一标识一个区块，没有歧义，任何节点通过对区块头进行简单哈希计算就可以独立得到标识。

需要注意的是，区块哈希并没有包含在区块的数据结构中，既不会在区块传输时存在，也不会作为区块链的一部分保存到节点的持久化存储设备中。实际上，区块哈希只在节点从网络上接收到区块时才自行计算生成。区块哈希可以保存在一个独立的数据库表中，作为区块元数据的一部分，以便索引和从磁盘上快速存取区块。

标识区块的另一种方式是它在区块链中的位置，被称为区块高度（**block height**）。第一个被创建的区块高度为0，它与前面提到的区块哈希为000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f的区块是一样的。因此，一个区块可以通过两种方式标识：引用区块哈希或者引用区块高度。在区块链中，每个被加到上一区块顶部的后续区块，都要比前一区块“高”一个位置，就像堆叠到其他盒子上面的盒子。2014年1月1日的区块高度大约是278000，意味着总共有278000个区块叠加到了2009年1月创建的第一个区块之上。

不像区块哈希，区块高度并不是唯一的标识符。虽然单一的区块总

有个特定的不变高度，但是反之则不然——区块高度不总是能标识一个单一区块。有可能有多个区块同时拥有相同高度，共同参与竞争区块链中的同一位置。这种情形将在第8章“区块链分叉”中讨论。区块高度也不是区块数据结构的一部分，它也没有存储在区块上。每个节点从网络上接收到区块时，在区块链中动态标识区块的位置（高度）。区块高度也可以作为元数据存储在一个索引数据库中，以提高存取速度。

一个区块的区块哈希总能唯一标识一个区块。一个区块也总是具有一个特定的区块高度。但是特定的区块高度却不一定能唯一标识一个区块。实际上，可能两个或多个区块会参与竞争区块链中的同一个位置。

创世区块

区块链中的第一个区块叫作创世区块，创建于2009年。它是区块链中所有区块的共同祖先，也就是说，如果从任何一个区块开始，沿着区块链回溯，最终都会到达创世区块。

每个节点启动时，其区块链中至少包含一个区块，因为创世区块是被静态编码到比特币客户端软件中的，无法被修改。每个节点都“知道”创世区块的哈希和结构、其创建时间，以及它包含的唯一一个交易。这样每个区块都拥有了区块链的起始点，一个安全的“根”，从它开始就可以构建一条可信任的区块链。

若需了解比特币核心客户端是如何静态编码创世区块的，可以查看chainparams.cpp源代码（**http://bit.ly/1xbrcwp**）。

以下标识哈希属于创世区块：
`000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f`

你也可以使用任何区块链浏览器网站（比如：blockchain.info），通过提供包含这个哈希值的URL，搜索区块链上的这个区块，你将看到一个页面描述这个区块的内容：
https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

https://blockexplorer.com/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

也可以使用比特币核心标准客户端的命令行：

```
$ bitcoind getblock
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

{
    "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
    "confirmations" : 308321,
    "size" : 285,
    "height" : 0,
    "version" : 1,
    "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afde
da33b",
    "tx" : [
        "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
    ],
    "time" : 1231006505,
    "nonce" : 2083236893,
    "bits" : "1d00ffff",
    "difficulty" : 1.00000000,
    "nextblockhash" :
"00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
}
```

创世区块中隐含着一条消息。其铸币交易的输入包含一段话："The Times 03/Jan/2009 Chancellor on brink of second bailout for banks".（《泰晤士报》，2009年1月3日，财政大臣正处于实施第二轮银行紧急援助的边缘。）这段话通过引用英国报纸《泰晤士报》（*The Times*）的头条新闻，证明了这个区块的创建日期（不早于2009年1月3日）。比特币诞生时，一场空前的货币危机正席卷全球，这段话以半开玩笑的方式提醒建立一种独立货币系统的重要性。这段话被比特币的创建者中本聪嵌入到第一个区块中。

区块链中连接区块

比特币完全节点从创世区块开始，维护着一套完整的区块链副本。随着新区块的发现，区块链的本地副本用于延展链条而得到持续更新。当节点从网络上接收到传入的区块时，首先对其进行验证，通过验证后，节点会检查其区块头，查找“前序区块哈希”，并使用该值与已存在的区块链进行连接。

我们来做个假设，一个节点在本地区块链副本中共有277314个区块。那么，这个节点所知的最新区块就是区块277314，其区块头哈希为0000000000000002 7e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249。

比特币节点从网络上接收到一个新区块，经过解析，看起来像这样：

```
{
    "size" : 43560,
    "version" : 2,

  "previousblockhash" :
      "0000000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",
  "merkleroot" :
      "5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",
  "time" : 1388185038,
  "difficulty" : 1180923195.25802612,
  "nonce" : 4215469401,
  "tx" : [
      "257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",

 #[... many more transactions omitted ...]

      "05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"
  ]
}
```

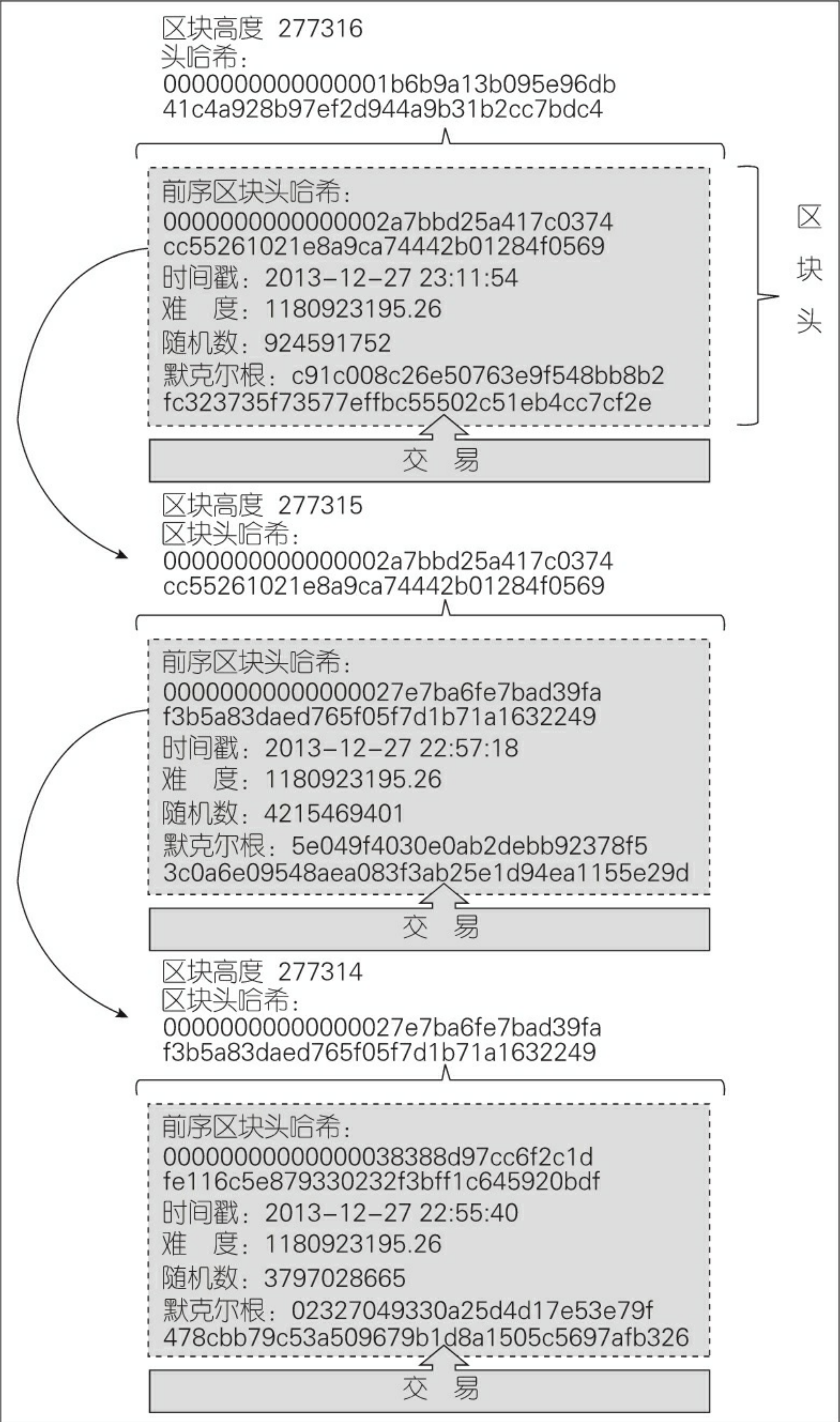
从新区块中找到previousblockhash字段，这个字段包含了它的父区块的哈希。这个哈希值对节点来说是已知的，就是在高度277314上的区块。这样，这个新区块就成了链上最后一个区块的子区块，区块链的长度延伸了，高度变为277315。图7.1是一条由3个区块组成的区块链，通过引用previousblockhash字段进行连接。

默克尔树

区块链中的每个区块都使用默克尔树来代表区块中所有交易的摘要。

默克尔树，也被称为二叉哈希树（**binary hash tree**），用于高效汇总和验证大数据集的完整性。默克尔树是一个由加密哈希组成的二叉树。名词“树”在计算机科学中用于描述分支的数据结构。作为数据结构，这里的“树”与通常理解的不同，在示意图上它的“根”常常被放在最上面，而“叶子”则在底部，形成一个自上而下的结构。在下面的例子中将看到一个示例图。

在比特币中，默克尔树通过产生一个全部交易集的数字指纹，以汇总区块中的所有交易，从而提供了一套非常高效的流程，以验证交易是否包含在区块里。通过递归计算一对对节点的哈希值，直到只剩一个节点，即根或默克尔根，就构成了一棵默克尔树。比特币中默克尔树采用的加密哈希算法是SHA256，因为需要重复两次，所以也称之为双重SHA256。



区块高度 277316
头哈希：
0000000000000001b6b9a13b095e96db
41c4a928b97ef2d944a9b31b2cc7bdc4
前序区块头哈希：
0000000000000002a7bbd25a417c0374
cc55261021e8a9ca74442b01284f0569
时间戳：2013-12-27 23:11:54
难　度：1180923195.26
随机数：924591752
默克尔根：c91c008c26e50763e9f548bb8b2
fc323735f73577effbc55502c51eb4cc7cf2e
区块头
交　易
区块高度 277315
区块头哈希：
0000000000000002a7bbd25a417c0374
cc55261021e8a9ca74442b01284f0569
前序区块头哈希：
00000000000000027e7ba6fe7bad39fa
f3b5a83daed765f05f7d1b71a1632249
时间戳：2013-12-27 22:57:18
难　度：1180923195.26
随机数：4215469401
默克尔根：5e049f4030e0ab2debb92378f5
3c0a6e09548aea083f3ab25e1d94ea1155e29d
交　易
区块高度 277314
区块头哈希：
00000000000000027e7ba6fe7bad39fa
f3b5a83daed765f05f7d1b71a1632249
前序区块头哈希：
0000000000000000038388d97cc6f2c1d
fe116c5e879330232f3bff1c645920bdf
时间戳：2013-12-27 22:55:40
难　度：1180923195.26
随机数：3797028665
默克尔根：02327049330a25d4d17e53e79f
478cbb79c53a509679b1d8a1505c5697afb326
交　易

图7.1　通过引用前序区块头哈希，将区块连接成一条链

当N个数据元素被哈希并汇总到一棵默克尔树，你就可以检查某个元素是否已被包含在树上，这个检查过程只需进行最多$2\times\log_2(N)$次运算，可以看出这种数据结构是非常高效的。

默克尔树自底部至上进行构建。在后面的例子中，我们将创建4个交易的默克尔树，A、B、C、D是构成默克尔树的叶子节点，如图7.2所示。交易数据本身不存储于默克尔树上，实际上，保存在叶子节点中的数据是交易数据经过哈希计算的结果，记为H_A,H_B,H_C和H_D。

```
H~A~ = SHA256(SHA256(Transaction A))
```

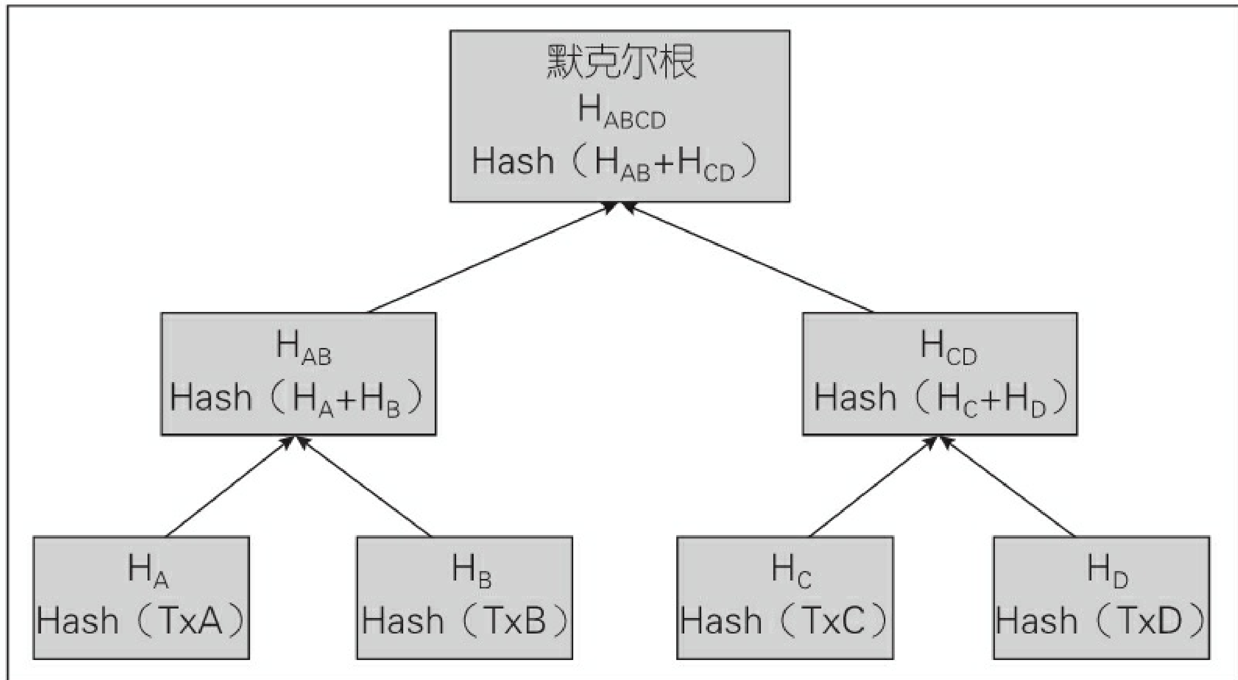


图7.2　在默克尔树上计算节点

通过连接两个哈希值，并进行哈希计算，相邻的叶子节点对得以在父节点上汇总。比方说，为构建一个父节点H_{AB}，两个子节点的32字节哈希值被连接在一起，形成一个64字节的字符串。接着，这个字符串进行双重哈希，构成父节点的哈希。

```
H~AB~ = SHA256(SHA256(H~A~ + H~B~))
```

过程持续进行，直到只在顶端留下一个节点，这个节点就是默克尔根。这个32字节哈希值存储于区块头中，作为所有4个交易数据的汇总。

由于默克尔树是一个二叉树，它需要偶数数量的叶子节点。如果只有奇数数量的交易需要进行汇总，那么需要复制最后一个交易的哈希，作为一个新的叶子节点，保证叶子数量为偶数，即形成平衡树（**balanced tree**）。如图7.3所示，交易C就被复制了。

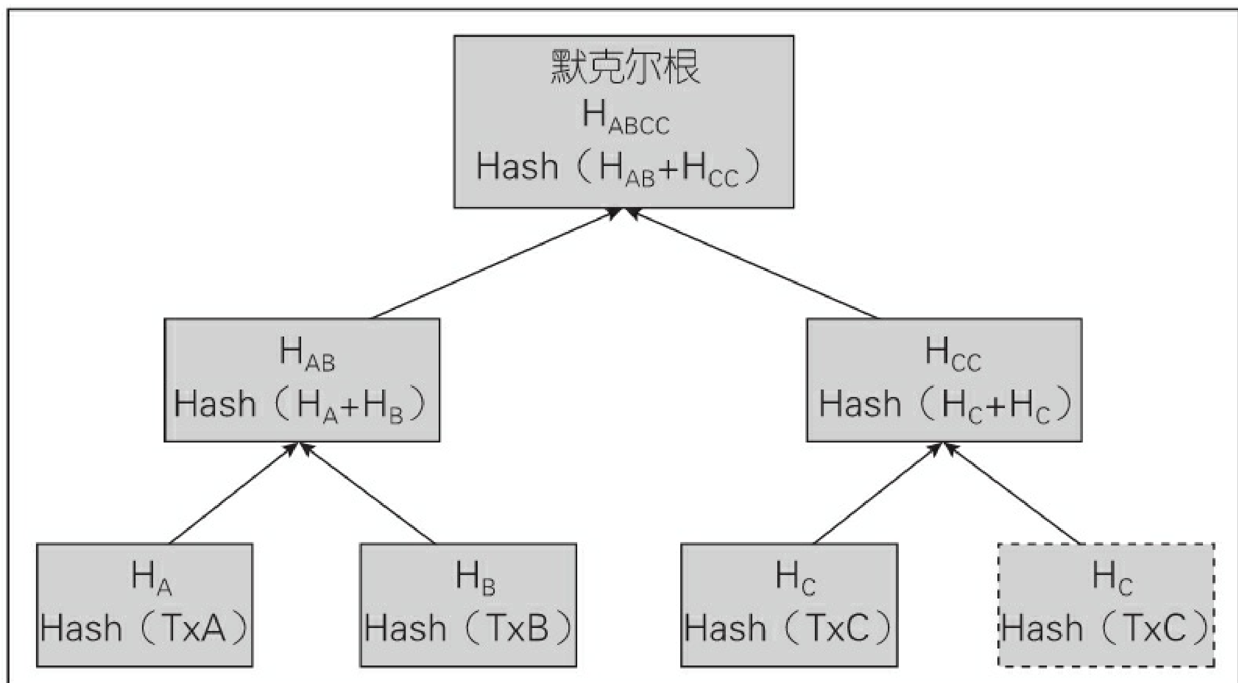


图7.3　复制一个数据元素以得到偶数数量的数据元素

使用同样的方法，可以创建任何尺寸的默克尔树。在比特币中，一个区块中含有几百上千个交易是很平常的，它们构建交易汇总的方法与前述方法完全一样，最终都会产生一个32字节的数据作为默克尔树根。在图7.4中，你将看到一棵由16个交易构成的树。请注意，虽然根在图上看起来比叶子节点大很多，实际上它们的尺寸完全一样，都是32字节。不管区块中是1个交易还是100个交易，默克尔根总是将它们汇总成

32字节。

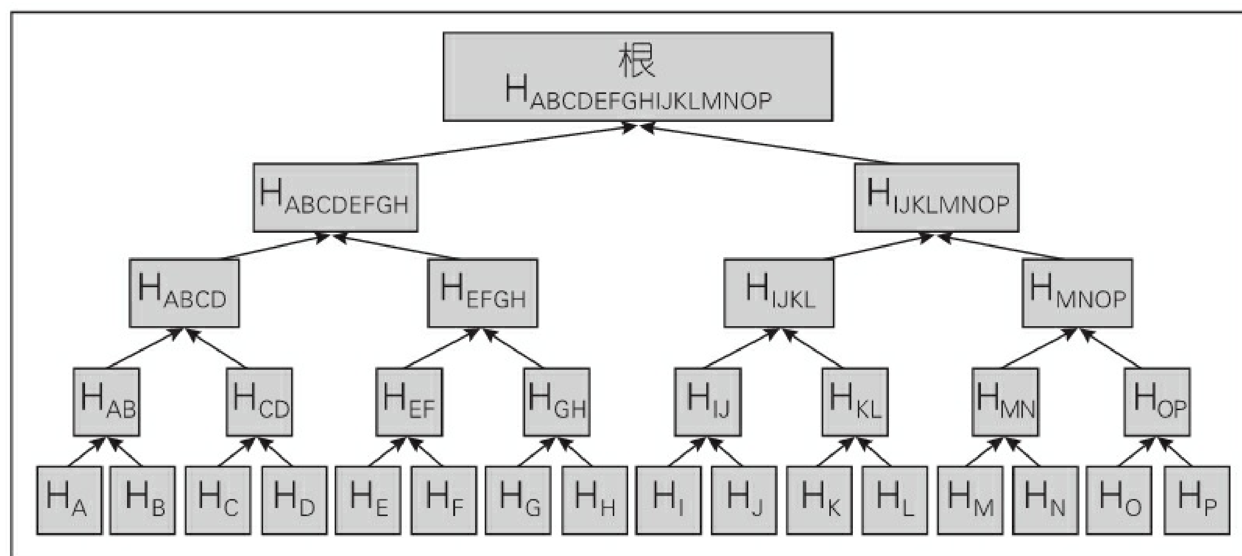


图7.4　一棵汇总了很多数据元素的默克尔树

为了证明一个特定交易包含在区块中，节点只需创建$\log_2$（N）个32字节的哈希值，形成一条从交易到根的路径，叫作认证路径或默克尔路径。当交易数量增长时，这显得尤为重要，因为底数为2的交易数量的对数增长，相比交易数量的增长要慢得多。这允许比特币节点高效地产生一条10到12个哈希值（320到384字节）的路径，从而证明交易是否属于区块，通常一个1M左右大小的区块含有超过1000个交易。

在图7.5中，节点只要产生一条包含4个哈希的默克尔路径，就能够证明交易K从属于区块，每个哈希32字节，总共128字节。这条路径包含4个哈希值：H_L、H_{IJ}、H_{MNOP}和$H_{ABCDEFGH}$。这4个哈希组成了一条认证路径，再加上另外4个与这些节点成对出现的哈希值H_{KL}、H_{IJKL}、$H_{IJKLMNOP}$，以及默克尔根，任何节点都可通过计算证明H_K包含在默克尔根中。

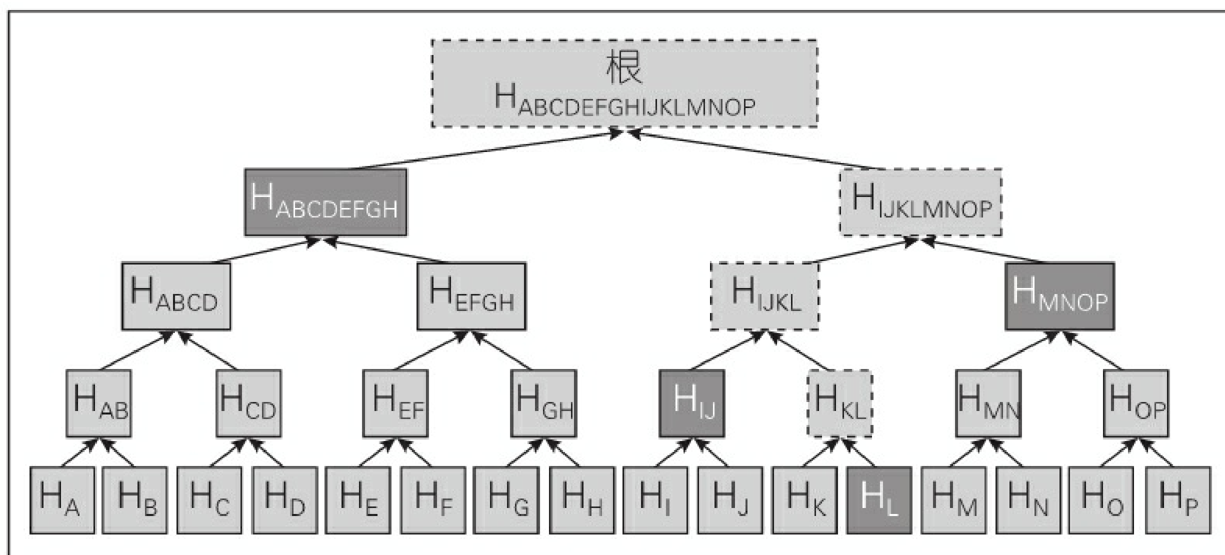


图7.5　一条默克尔路径，用于证明包含某个数据元素

例7-1的代码演示了从叶子节点哈希一直到根节点、创建默克尔树的过程，例子中使用到了libbitcoin库中的一些辅助函数。

例**7-1**　创建一棵默克尔树

```
#include <bitcoin/bitcoin.hpp>

bc::hash_digest create_merkle(bc::hash_digest_list& merkle)
{
    // Stop if hash list is empty.
    if (merkle.empty())
        return bc::null_hash;
    else if (merkle.size() == 1)
        return merkle[0];

    // While there is more than 1 hash in the list, keep looping...
    while (merkle.size() > 1)
    {
```

```
    // If number of hashes is odd, duplicate last hash in the list.
    if (merkle.size() % 2 != 0)
        merkle.push_back(merkle.back());
    // List size is now even.
    assert(merkle.size() % 2 == 0);

    // New hash list.
    bc::hash_digest_list new_merkle;
    // Loop through hashes 2 at a time.
    for (auto it = merkle.begin(); it != merkle.end(); it += 2)
    {
        // Join both current hashes together (concatenate).
        bc::data_chunk concat_data(bc::hash_size * 2);
        auto concat = bc::make_serializer(concat_data.begin());
        concat.write_hash(*it);
        concat.write_hash(*(it + 1));
        assert(concat.iterator() == concat_data.end());
        // Hash both of the hashes.
        bc::hash_digest new_root = bc::bitcoin_hash(concat_data);
        // Add this to the new list.
        new_merkle.push_back(new_root);
    }
    // This is the new list.
    merkle = new_merkle;

    // DEBUG output -------------------------------------
    std::cout << "Current merkle hash list:" << std::endl;
    for (const auto& hash: merkle)
        std::cout << "  " << bc::encode_hex(hash) << std::endl;
    std::cout << std::endl;
    // --------------------------------------------------
}
```

```
    // Finally we end up with a single item.
    return merkle[0];
}

int main()
{
    // Replace these hashes with ones from a block to reproduce the same merkle
root.
    bc::hash_digest_list tx_hashes{{
        bc::de
code_hash("0000000000000000000000000000000000000000000000000000000000000000"),
        bc::de
code_hash("0000000000000000000000000000000000000000000000000000000000000011"),
        bc::de
code_hash("0000000000000000000000000000000000000000000000000000000000000022"),
    }};
    const bc::hash_digest merkle_root = create_merkle(tx_hashes);
    std::cout << "Result: " << bc::encode_hex(merkle_root) << std::endl;
    return 0;
}
```

例7-2显示了编译运行代码的结果。

例**7-2**　编译并运行示例代码

```
$ # Compile the merkle.cpp code
$ g++ -o merkle merkle.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the merkle executable
$ ./merkle
Current merkle hash list:
  32650049a0418e4380db0af81788635d8b65424d397170b8499cdc28c4d27006
  30861db96905c8dc8b99398ca1cd5bd5b84ac3264a4e1b3e65afa1bcee7540c4

Current merkle hash list:
  d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

Result: d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3
```

当规模增长时，默克尔树的效率也变得非常明显。表7.3展示了为证明区块中存在某交易而创建默克尔路径所需交换的数据量。

表7.3　默克尔树的效率

交易数量（个）	区块粗略大小（字节）	路径尺寸（哈希数）	路径尺寸（字节）
16	4K	4	128
512	128K	9	288
2 048	512K	11	352
65 535	16M	16	512

正如表7.3中所看到的，区块的大小增长很快，从4KB、16个交易，到16MB、65535个交易，但是用于证明交易是否存在的默克尔路径增长却慢得多，仅仅从128字节增长到512字节。有了默克尔树，节点可以只下载区块头（每区块80字节），通过从其他完全节点获取一个很小的默克尔路径，即可以判断交易是否包含在区块中，不需要存储或传输区块链中的绝大部分内容，这些数据有好几十GB。不维护全量区块链的节点叫作简化支付验证节点，它们使用默克尔路径来验证交易，不需要下载全部区块。

默克尔树和简化支付验证

默克尔树在SPV节点中得到了广泛应用。SPV节点不保存全部交易，也不保存完整区块数据，仅仅存有区块头信息。在不必下载区块中完整交易的情况下，SPV节点利用认证路径或默克尔路径来验证交易是否包含在区块中。

举例来看，假设一个SPV节点，它对接收到的、向它钱包软件中某个地址进行支付的交易感兴趣。SPV节点在与其他对等节点的连接上建立一个布隆过滤器，限制只接收那些与它感兴趣的地址相关的交易。当对等节点看到某个交易与布隆过滤器匹配时，就使用merkleblock消息将相关区块发送到SPV节点上。merkleblock消息包含区块头以及一条将区块中感兴趣的交易连接到默克尔根的默克尔路径。SPV节点可利用默克尔路径将交易与区块相连，并验证交易确实被区块所包含。SPV节点也利用区块头信息将区块连接到区块链。交易与区块、区块与区块链，这两个连接的组合，提供了交易已被记录于区块链中的证明。总之，SPV节点收到的包含区块头和默克尔路径的数据还不到1KB，比整个区块数据（目前大约1MB）小了1000倍。

第8章 挖矿与共识

介绍

挖矿是维持比特币货币供应的一个过程。同时，挖矿也保护着比特币系统的安全，防止欺诈交易或者在不同交易里使用同一笔比特币资金，即双重支付。矿工向比特币网络提供处理能力，以交换获取比特币奖励的机会。

矿工验证新的交易，并把它们记录到全局账本上。每隔10分钟左右，一个包含上个区块产生以来发生的所有交易的新区块就会被矿工“挖”出，经过挖矿，这些新交易记录成为区块链的一部分。成为区块一部分并被加入区块链中的交易是“已确认的”交易，比特币的新所有者可以花费在这些交易中收到的比特币。

矿工们获得两种类型的挖矿奖赏：一种是每个新区块中产生的新比特币；另一种是新区块中包含的所有交易的交易费用。为了赢得这些报酬，矿工通过竞争的方式基于加密哈希算法解决一个极为复杂的数学问题。问题的解被称为工作量证明，包含在新区块中，作为矿工所付出的计算工作量的证明。竞争解决工作量证明算法、赢取奖励，以及在区块链上记录交易的权利，构成了比特币安全模型的基础。

新比特币产生的过程之所以叫作挖矿，是因为其奖励机制是模拟收益递减的，就像贵金属矿产的挖掘工作，越挖越少。比特币的货币供应通过挖矿来实现，类似中央银行通过印钞来发行货币。矿工可以加入区块的新比特币数量大概4年（精确地说是每210000个区块）就会减少一次。2009年1月，比特币网络刚开始运行的时候，每挖出一个区块有50比特币产生；到2012年11月，这个数额就减少一半，降到25比特币。到2016年7月（译者：原书为2016年某个时候所写），已降至12.5比特币。基于这个公式，比特币挖矿奖励以指数级下降，大约到2140年，所

有比特币（2099.999998万）都将被发行完毕。2140年后将不会再发行新比特币。

比特币矿工也可以从交易中赚取交易费。每个交易都可能包含有交易费用，交易费用以交易输入与交易输出之间的差值的形式存在。赢得竞争的比特币矿工获得区块中所有交易的“小费”。目前，交易费用仅占到矿工收入的0.5%甚至更少，矿工主要的收入来源还是新挖出的比特币。但随着时间推移，挖矿的奖励金额不断减少，而每个区块中所包含的交易却在增加，交易费用在矿工收入中的占比必将逐步增加。2140年后，由于新比特币枯竭，比特币矿工的收入将全部来自交易费用。

“挖矿”这个词很容易让人联想到贵金属挖掘，它将我们的注意力集中到挖矿的奖励，也就是每个区块产生的新比特币上。虽然挖矿的动作是被这些新比特币激励产生的，挖矿最主要的目的却不是获得报酬或者产生新比特币。如果仅仅将挖矿看作一个创建比特币的过程，你就把手段（激励）当成了此过程的目的。挖矿是虚拟的去中心化清算机构的主要工作过程，通过这个虚拟清算机构，交易得以验证和清算。挖矿保护了比特币系统的安全，使得在没有中心机构的情况下，全网能够形成共识。

挖矿的发明使比特币成为一种特别的货币，而去中心化的安全机制则构成了点对点数字货币的基础。铸币奖励以及交易费用作为一种激励方案，不仅规范了矿工的行为，使其符合网络安全的要求，同时也实现了货币的供应。

在本章中，我们首先研究作为货币供应机制的挖矿过程，然后将目光聚焦到其最重要的功能，即支撑了比特币安全的去中心化共识机制。

比特币经济学和货币铸造

比特币在创建区块的过程中，以固定和递减的速率被“铸造”出来。通常每10分钟左右产生一个新的区块，而每个区块均会从无到有产生全新的比特币。每经过210000个区块，或者大约4年，货币发行速率就会降低50%。比特币刚开始运作的首个4年，每个区块包含有50新比特币。

2012年11月，比特币的发行速率降低到了每区块25比特币，这个过程将继续下去；到2016年7月，发行速率将降低到每区块12.5比特币。新比特币的创建速率以这种指数级的方式递减，经过64次“减半”，直到13230000号区块被挖出（大致在2137年），每区块创建比特币的数量将会降到货币最小单位1聪。最终，大约到2140年，13440万个区块被创建后，所有2099999997690000聪，或者大约2100万比特币将全部发行完毕。从此以后，区块不再包含新比特币，矿工的报酬将完全来自交易费用。图8.1显示了随着货币发行量的递减，比特币总量与时间的对应关系。

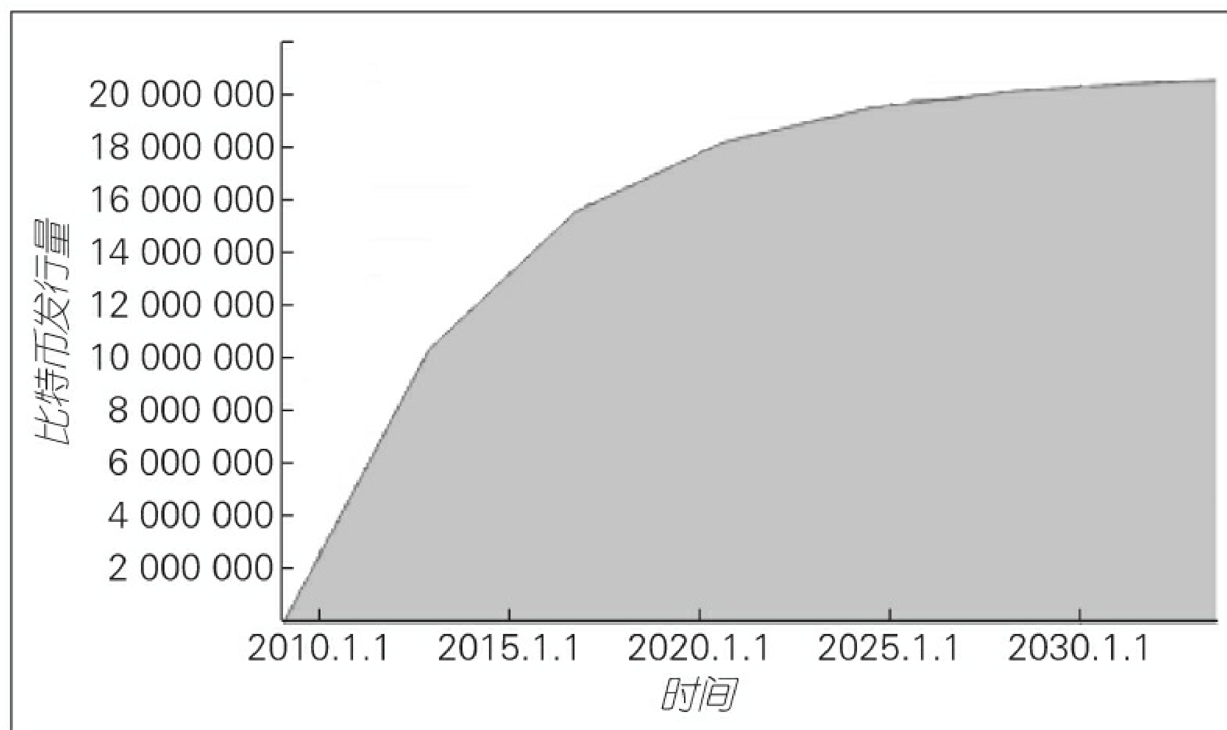


图8.1　在发行速率呈指数递减下，比特币货币供应与时间的关系

在示例8-1中，我们计算一下将被发行的所有比特币数量。

例**8-1**　计算总共会发行多少比特币的脚本

```
# Original block reward for miners was 50 BTC
start_block_reward = 50
# 210000 is around every 4 years with a 10 minute block interval
reward_interval = 210000

def max_money():
    # 50 BTC = 50 0000 0000 Satoshis
    current_reward = 50 * 10**8
    total = 0
    while current_reward > 0:
        total += reward_interval * current_reward
        current_reward /= 2
    return total

print "Total BTC to ever be created:", max_money(), "Satoshis"
```

例8-2展示了脚本运行后的输出结果。

例**8-2**　运行**max_money.py**脚本

```
$ python max_money.py
Total BTC to ever be created: 2099999997690000 Satoshis
```

总量有限和递减的发行速率保证了货币供应的稳定性，防止了通货膨胀的发生。不像中央银行可以无限制印制的法币，比特币永远不可能因为超发而导致通货膨胀。

通货紧缩的货币

固定且递减的货币发行机制造成的最重要也最具争议的结果，就是它将天然趋向于通货紧缩。通货紧缩是一种价值升值的现象，因为供应与需求的错配推高了货币的价值（以及兑换汇率）。与通货膨胀相反，

价格缩水意味着货币随着时间的推移拥有了更强的购买力。

很多经济学家认为通货紧缩经济是一场灾难，应不惜代价加以规避。因为一段时间内的快速通货紧缩，国民将产生价格会继续下降的预期，因此倾向将钱存起来而不是消费。这种现象在日本“失去的十年”期间得以无情地展现，当需求完全瓦解时，也将货币推入了通货紧缩的漩涡。

比特币专家认为通货紧缩本身并不是坏事。当然，通货紧缩是与需求的萎缩相关的，这也是我们唯一需要研究的有关通货紧缩的案例。在一个存在无限发行可能性的法定货币系统中，除非消费需求完全萎缩并且没有印钞意愿，陷入通货紧缩的漩涡是很困难的。比特币中的通货紧缩不是由消费需求萎缩引起的，而是由可预期的限制货币供应引起的。

在实践中，很明显的事实是，通货紧缩导致的货币囤积可以自发地被供应商的折扣所抵消，当折扣率达到一定程度，就可以战胜消费者的储蓄本能。因为不管商家还是消费者，都有囤积货币的动机，通过折扣最终将达成价格的平衡。在这个价格上，双方的囤积欲望互相匹配达到平衡。如果折扣达到30%，大多数基于比特币的零售商都可以轻松地重新挑起消费者的购物欲望，并获得收益。当然，比特币这种不是因经济快速衰退而引起的通货紧缩，是否还会引发其他问题，仍有待观察。

去中心化共识

在前面的章节中，我们观察了区块链——一个包含全部交易的全局公共账本（列表），比特币网络中的所有参与者都可以接受它，并将其视为所有权证明的权威记录。

但是，在不信任其他人的情况下，如何让网络中的每个参与者都能对一个关于谁拥有什么的普遍“真理”取得共识呢？所有的传统支付系统所依赖的信用模型都有一个提供清算服务的中央权威机构，对每一笔交易进行验证并进行清算处理。比特币没有中央权威机构，但是每个完全节点都有一份公共账本的完整复制，可以认为这是一份权威的记录。区块链不是由中央机构创建的，而是由网络中的每个节点独立组装而成。通过某种方式，网络上的每个节点，对在不安全的网络连接上传输的信息，可以达成一个共同的结论，并且能装配一份与别人完全相同的公共账本。本章将研究在没有中央机构介入的情况下，比特币如何达成全局共识的过程。

中本聪的发明主要在于建立了一种去中心化的自发共识（**emergent consesus**）机制。自发，是因为共识不是事先明确达成的——没有选举，也没有一个固定的达成共识的时刻。共识是自然产生的，是成千上万遵循共同的简单规则的节点，在异步交互过程中形成的。所有比特币的属性，包括货币、交易、支付，以及不依赖于中央机构或信用体系的安全模型，都从这一发明衍生而来。

比特币的去中心化共识是由4个过程的相互影响而自发产生的，这些过程是在网络上的节点中独立进行的。

●基于规则的完整列表，各个完全节点独立验证每个交易。

●通过基于工作量证明算法的证明运算，挖矿节点独立将交易汇聚到新区块中。

●每个节点独立验证新区块并将其整合进区块链。

●每个节点独立选择累积进行了最多工作量证明计算的链条。

在接下来的几个小节中，我们将考察这些过程，了解它们是如何通过互相作用，形成自发的全网共识，从而使任意节点组合出各自权威、可信、公开的总账。

独立交易验证

比特币共识机制的第一个步骤是各个节点独立验证每个交易。在第5章中，我们研究了钱包软件通过收集UTXO，提供合适的解锁脚本，创建指派给新所有者的输出，从而创建新交易的过程。通过以上过程，新建的交易被发送到网络中的邻居节点，并传播到整个比特币网络。

但是，将交易转发给邻居前，接收到交易的比特币节点首先会验证它的有效性。这使得只有有效的交易才会在网络中传播，而无效的交易在第一个接收到的节点就被丢弃了。

每个节点都要遵守一个很长的规则列表来验证交易的有效性。

●交易的语法和数据结构必须正确。

●交易的输入和输出均不能为空。

●交易字节数的大小必须小于MAX_BLOCK_SIZE。

●每个交易输出的汇总价值必须在允许范围内（小于2100万比特币，大于0）。

●任何交易输入的哈希不能为0，N不能等于-1（也就是铸币交易不能被转发）。

●nLockTime小于或等于INT_MAX。

●交易字节数必须大于或等于100。

●交易中签名操作的数量必须小于签名操作的限制值。

●解锁脚本（scriptSig）只能将数字压入堆栈，锁定脚本（scriptPubKey）必须匹配isStandard格式（这将拒绝“非标准”交易）。

●交易池或者主分支的区块中必须存在匹配的交易。

●对于每个输入，如果引用的输出在交易池的其他交易中存在，交易必须被拒绝。

●对于每个输入，需要在主分支和交易池中查找被引用的输出交易。如果任何输入对应的输出交易不存在，那么这就是个孤儿交易。如果其对应的交易不在孤儿交易池中，将其加入孤儿交易池。

●对于每个输入，如果引用的输出交易是一个铸币交易的输出，必须至少经过COINBASE_MATURITY（100）确认。

●使用输出交易计算输入价值，检查每个输入价值以及汇总值，看其是否超过允许范围（小于2100万比特币，大于0）。

●如果输入价值汇总小于输出价值，拒绝这笔交易。

●如果交易费用太小以致无法加入一个空的区块，拒绝这笔交易。

●每个输入的解锁脚本必须基于相应的输出锁定脚本进行验证。

这些规则的细节可以在比特币标准客户端的函数AcceptToMemoryPool，CheckTransaction，CheckInputs中查到。需要注意的是，这些条件经常变动，比如添加新约束条件以防范新类型的拒绝服务攻击，或者放松某些规则以支持新类型的交易。

通过对每一个交易在接收后和传播前进行独立验证，每个节点都会创建一个有效新交易的池子（交易池），而不同节点间交易池内交易的

顺序也会大致相同。

挖矿节点

比特币网络上的某些特殊节点，称作“矿工”（miners）。在第1章中，我们介绍过景，一个在中国上海的计算机工程专业学生，他就是一个比特币矿工。景通过运行一套为比特币挖矿定制的计算机硬件系统（叫作“矿机”）来赚取比特币。景的定制挖矿硬件连接到一台运行完全节点的服务器上。其他一些矿工则与景不同，他们是在没有完全节点的情况下进行挖矿的，我们将在本章的“矿池”中介绍。跟其他完全节点一样，景的节点也在比特币网络上接收并传播未确认交易。当然，景的节点同时还将这些交易整合到新区块中。

景的节点随时监听新的区块，并将其传播到网络中，就像所有其他节点一样。但是，新区块的到来对挖矿节点来说有着特殊的意义。矿工间的竞争随着新区块的传播而终止，新区块是这次竞争中最终赢家的胜利宣言。对于矿工来说，接收到一个新区块意味着有人已经赢得了这次竞争，而其他人输了。但是一轮竞争的结束也同样意味着新一轮竞争的开始。新区块不仅是一面方格旗，标志着一轮竞赛的结束；它也是发令枪，标志着下一个区块的竞赛开始了。

整合交易到区块中

验证交易之后，比特币节点会将它们加入内存池（或交易池）中，交易在那里等待着被加入（挖矿）一个区块内，这是比特币共识机制的第二个步骤。景的节点与其他节点一样，收集、验证并转发交易。与其他节点不同的是，景的节点在完成这些动作后，还要把这些交易整合到一个候选区块（**candidate block**）中。

我们来跟踪一些区块，这些区块是在爱丽丝从鲍勃咖啡店购买一杯咖啡期间创建的（参见第2章中“购买一杯咖啡”）。爱丽丝的交易被包含进277316号区块。为了演示本章所阐述的概念，我们假设这个区块是被景的挖矿系统挖出来的，我们将在爱丽丝的交易成为新区块的一部分后，继续跟进这笔交易。

景的挖矿节点维护着一份区块链的完整副本，它是所有区块的列表，包含了2009年比特币系统创建以来的所有区块。在爱丽丝购买那杯咖啡前，景的节点已经装配了一条包含277314个区块的链条。景的节点持续监听交易，尝试挖出新的区块；同时，它也监听其他节点发现的区块。当景的节点正在挖矿时，它从比特币网络中接收到277315号区块。这个区块的到达，标志着区块277315的挖矿竞争结束了，而创建区块277316的竞争从此开始。

在之前的10分钟内，景的节点在查找区块255315的解决方案的过程中，也在收集交易，为创建下个区块做准备。此时，它已经收集了几百个交易并保存在内存池中。当接收到区块277315并验证后，景的节点对内存池中的交易进行检查，剔除那些已被包含进277315区块的交易。仍然留在内存池中的交易都是未确认的交易，它们继续等待被记录到新区块中。

完成这些准备工作后，景的节点立即创建一个新的空区块，作为区块277316的候选，这个区块就叫作候选区块，因为它尚未成为有效区块，不包含有效的工作量证明。只有在矿工成功找到一个工作量证明算法的解后，这个区块才会变为有效。

交易年龄、费用和优先级

为了构建候选区块，景的比特币节点需要从内存池中选择交易。选择过程首先要对每个交易赋予一个优先级权数，并将最高优先级的交易优先选出。交易基于输入中即将被花费的UTXO的“年龄”进行排序，允许那些较老的、高价值的交易输入比那些较新的、低价值的交易输入拥有更高的优先级。只要区块空间足够，高优先级的交易可以免费发送。

交易优先级是通过输入价值与输入“年龄”乘积的汇总再除以交易总大小后得出的。

```
Priority = Sum (Value of input * Input Age) / Transaction Size
```

在等式中，输入价值的单位是比特币的基础单位，即聪（1比特币的1亿分之一）。UTXO的年龄是自UTXO被记入区块起所经过的区块数量，即这个UTXO在区块链中的深度。交易记录的大小用字节来表示。

对一个被认定为“高优先级”的交易来说，它的优先级必须大于57600000，这相当于一个包含1比特币（1亿聪）、年龄为1天（144区块）、大小为250字节的交易。

```
High Priority > 100,000,000 satoshis * 144 blocks / 250 bytes = 57,600,000
```

区块交易空间的前50KB是保留给高优先级交易的。景的节点将首先填充前50KB，最高优先级的交易优先处理，不管有没有交易费用。即使没有交易费用，高优先级交易也能得到处理。

接着，景的节点继续填充区块的剩余部分，直到达到其大小的上限

（代码中设定的MAX_BLOCK_SIZE），这部分交易必须至少包含最低交易费用，并且依据每千字节交易费用的高低进行排序。

如果区块中仍有空间，景的节点可能会选择没有交易费用的交易来填充它。某些矿工基于最大努力原则，将没有交易费用的交易加入区块，而有些矿工则可能选择忽略那些没有交易费的交易。

区块填充后，若还有交易留在内存池中，它们将继续在内存池中等待下个区块的处理。由于交易停留在内存池中，它们的输入“年龄”，即它们花费的UTXO在区块链中的深度将随着新区块的加入而变得更深。因为交易的优先级基于输入年龄，交易保留在内存池中会“变老”，从而优先级得以提高。最终，没有交易费用的交易也有可能拥有足够高的优先级，并被免费添加进区块当中。

比特币的交易并没有超时的设置。一个当前有效的交易也将永久有效。但是，如果一个交易只在网络中传播一次，它只会停留在矿工节点的内存池中。由于内存是一种临时的、非持久化的存储形式，当矿工节点重启后，它的内存池就会被清空。虽然有效交易可能已经被传播到网络上，但是如果它一直未被处理，最终可能从所有挖矿节点的内存池中消失。如果交易未在一定时间内得到处理，钱包软件应该重新发送交易或重新创建包含较高交易费用的交易。

当景的节点汇聚了内存池中的所有交易后，新的候选区块总共填充了418个交易，合计交易费用为0.09094928比特币。你可以利用比特币核心的命令行接口看到区块链中的这个区块，如例8-3所示。

```
$ bitcoin-cli getblockhash
277316000000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4

$ bitcoin-cli getblock
0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4
```

例**8-3**　区块**277316**

```
{
    "hash" : "0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
    "confirmations" : 35561,
    "size" : 218629,
    "height" : 277316,
    "version" : 2,
    "merkleroot" :
"c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e",
    "tx" : [
        "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
        "b268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe",

        ... 417 more transactions ...
        ],
     "time" : 1388185914,
     "nonce" : 924591752,
     "bits" : "1903a30c",
     "difficulty" : 1180923195.25802612,
     "chainwork" :
"000000000000000000000000000000000000000000000934695e92aaf53afa1a",
     "previousblockhash" :
"0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569",
     "nextblockhash" :
"000000000000000010236c269dd6ed714dd5db39d36b33959079d78dfd431ba7"
}
```

铸币交易

加入区块的第一笔交易是一个特殊交易，叫作铸币交易（**generation transaction**）或币基交易（**coinbase transaction**）。这个交易由景的节点创建，也是对他挖矿付出努力的报酬。景的节点创建了一笔铸币交易，内容为向其钱包发起支付：“支付25.09094928比特币到景的地址。”景通过挖出一个区块获得的总报酬为币基奖励（25新比特币）与区块中所有交易的交易费用汇总（0.09094928比特币）的和，如例8-4所示。

```
$ bitcoin-cli getrawtransaction
d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f 1
```

例**8-4**　铸币交易

```
{
    "hex" :
"01000000010000000000000000000000000000000000000000000000000000000000000000ffffffff0f03443b0403858402062f503253482fffffffff0110c08d9500000000232102aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac00000000",
    "txid" : "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "coinbase" : "03443b0403858402062f503253482f",
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 25.09094928,
            "n" : 0,
            "scriptPubKey" : {
                "asm" :
"02aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21OP_CHECKSIG",
                "hex" :
"2102aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac",
                "reqSigs" : 1,
                "type" : "pubkey",
                "addresses" : [
                    "1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N"
                ]
            }
        }
    ],
    "blockhash" :
"0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
    "confirmations" : 35566,
    "time" : 1388185914,
    "blocktime" : 1388185914
}
```

不像普通交易，铸币交易并不需要消耗（花费）UTXO。实际上，它只有一个输入，叫作币基（**coinbase**），这个交易从无到有生成了比特币。铸币交易有一个输出，支付到矿工的比特币地址。铸币交易的输

出将25.09094928比特币发送到矿工的比特币地址。在本例中，地址为：1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N。

币基奖励与交易费用

首先，为了构建一笔铸币交易，景的节点对所有加入区块的418个交易的输入和输出进行汇总轧差，计算得出交易费用。计算公式如下。

```
Total Fees = Sum(Inputs) - Sum(Outputs)
```

在区块277316中，总的交易费用是0.09094928比特币。

接着，景的节点需要计算新区块的准确奖励金额。奖励金额的计算基于区块高度，从每区块50比特币开始，每210000个区块减半。当前的区块高度是277316，因此正确的奖励是25比特币。

计算过程可以在比特币核心客户端的函数GetBlockValue中查到，如例8-5所示。

例**8-5**　计算区块奖励——函数**GetBlockValue**，比特币核心客户端，**main.cpp**，第**1305**行

```
int64_t GetBlockValue(int nHeight, int64_t nFees)
{
    int64_t nSubsidy = 50 * COIN;
    int halvings = nHeight / Params().SubsidyHalvingInterval();

    // Force block reward to zero when right shift is undefined.
    if (halvings >= 64)
```

```
        return nFees;

    // Subsidy is cut in half every 210,000 blocks which will occur approximately
every 4 years.
    nSubsidy >>= halvings;

    return nSubsidy + nFees;
}
```

初始奖励是以聪为单位进行计算的，其值为50与COIN常量（100000000聪）的乘积。即初始的奖励金（nSubsidy）为50亿聪。

接下来，计算已经发生的减半（halvings）次数：将当前的区块高度除以减半间隔（SubsidyHalvingInterval）。对区块277316来说，除以210000的减半间隔，其结果为1，即1个减半。

可允许的最大的减半次数为64次，所以在代码中，如果减半次数超出了64，就将奖励金设置为0（只返回交易费用）。

再接下来，函数采用右移操作符对奖励金（nSubsidy）进行除以2的操作，每次减半右移一位，即除以2。对于区块277316，由于减半次数为1次，则对奖励金（50亿聪）右移操作一次，得到的结果是25亿聪，或者25比特币。之所以使用右移操作符，是因为它做除以2的操作效率比整数除法或浮点型除法高得多。

最后，函数将币基奖励（nSubsidy）与交易费用（nFees）相加，将两者总和返回。

铸币交易的结构

通过以上计算，景的节点创建了一个铸币交易，向他自己支付了25.09094928比特币。

从例8-4可以看到，铸币交易使用了一种特殊的格式。相对普通交易的输入需要指定用于花费的前序UTXO，铸币交易只有一个“币基”输入。我们曾在表5.3中考察了普通交易的输入。现在我们将普通交易的输入与铸币交易的输入做个对比。表8.1显示的是普通交易输入的数据结构，表8.2显示的是铸币交易输入的数据结构。

表8.1　“普通”交易的输入结构

大小	字段	描述
32 字节	交易哈希（Transaction Hash）	指向待花费 UTXO 的指针
4 字节	输出索引（Output Index）	UTXO 的编号，从 0 开始
1 ~9 字节（VarInt）	解锁脚本大小（Unlocking-Script Size）	紧跟的解锁脚本长度
可变长度	解锁脚本（Unlocking-Script）	满足 UTXO 锁定脚本条件的解锁脚本
4 字节	序号（Sequence Number）	当前尚未启用的 Tx 替代功能，设置为 0xFFFFFFFF

（续表）

表8.2　铸币交易的输入结构

大小	字段	描述
32 字节	交易哈希（Transaction Hash）	所有位均为 0：不是一个交易哈希引用
4 字节	输出索引号（Output Index）	所有位均为 1：0xFFFFFFFF
1 ~9 字节（VarInt）	币基数据长度（Coinbase Data Size）	币基数据的长度，从 2 到 100 字节
可变长度	币基数据（Coinbase Data）	任意长度的数据，用于额外的随机数以及 v2 区块中的挖矿标签，必须以区块高度开头
4 字节	序列号（Sequence Number）	设置为 0xFFFFFFFF

在铸币交易中，前两个字段设置为与UTXO引用无关的值。第一个字段是32字节的“0”，而不是“交易哈希”。“输出索引”用4字节0xFF填充（十进制255）。“解锁脚本”被替换为币基数据，一个可由矿工自由定义的数据。

币基数据

铸币交易没有解锁脚本（scriptSig）字段。相反，这个字段被替换为币基数据，长度限定在2到100字节之间。除了前面几个字节，币基数据的剩余部分可被矿工用于其自主的任何用途，填充任意数据。

举例来说，在创世区块中，中本聪在币基数据中加了这段话：“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks”（《泰晤士报》，2009年1月3日，财政大臣正处于实施第二轮银行紧急援助的边缘）用以证明比特币的发明日期并传达一条信息。当前，矿工们通常使用币基数据包含额外的随机数，并附上标识其矿池信息的字符串，我们将在接下来的几个章节继续讨论。

币基的前几个字节曾经也是可以任意安排的，但是现在不再这样了。依据比特币改进提案34号（BIP0034），版本2区块（版本字段设置为2的区块）必须在币基字段的最前面附加区块高度索引，作为脚本的“压栈”操作。

在区块277316中，我们看到币基（参看例8-4），也就是交易输入的“解锁脚本”或scriptSig字段，包含一段十六进制数据03443b0403858402062f503253482f。我们将其解码，看看其内容。

第1个字节，03，指示脚本执行引擎将后续3个字节压入脚本堆栈中（参见附录A表A.1）。接下来的3个字节，0x443b04，以小字节序（little endian）格式编码的区块高度。将其字节序翻转，结果就是0x043b44，对应的十进制就是277316。

紧接着的几个十六进制数字（03858402062）用于编码额外随机数（参看本章中“扩展随机数方案”），以用于找到合适的工作量证明的解。

最后部分（2f503253482f）是ASCII编码的字符串（“/P2SH/”），提示本区块的挖矿节点支持BIP0016定义的“支付到脚本哈希（P2SH）”。P2SH能力引入的时候曾要求矿工“投票”，从BIP0016和BIP0017中间选择一个。那些选择了BIP0016实现的矿工会将“/P2SH/”加进币基数据，而那些选择了BIP0017的P2SH实现的矿工则在币基数据中加入字符串“p2sh/CHV”。最终BIP0016成了赢家，但是很多矿工依然将字符串/P2SH/加入币基中，表明其支持这个特性的态度。

例8-6使用libbitcoin库（参见第3章“替代客户端、库、工具集”）从创世区块中提取币基数据，并显示中本聪在区块中留下的信息。需要注意的是，libbitcoin内嵌了创世区块的静态复制，所以示例代码可以直接从库中提取创世区块。

例**8-6**　从创世区块提取币基数据

```
/*
  Display the genesis block message by Satoshi.
*/
#include <iostream>
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Create genesis block.
    bc::block_type block = bc::genesis_block();
    // Genesis block contains a single coinbase transaction.
    assert(block.transactions.size() == 1);
    // Get first transaction in block (coinbase).
    const bc::transaction_type& coinbase_tx = block.transactions[0];
    // Coinbase tx has a single input.
    assert(coinbase_tx.inputs.size() == 1);
    const bc::transaction_input_type& coinbase_input = coinbase_tx.inputs[0];
    // Convert the input script to its raw format.
    const bc::data_chunk& raw_message = save_script(coinbase_input.script);
    // Convert this to an std::string.
    std::string message;
    message.resize(raw_message.size());
    std::copy(raw_message.begin(), raw_message.end(), message.begin());
    // Display the genesis block message.
    std::cout << message << std::endl;
    return 0;
}
```

使用GNU　C++编译这段代码，运行所生成的可执行程序，结果如例8-7所示。

例**8-7**　编译运行**satoshi-words**示例代码

```
$ # Compile the code
$  g++ -o satoshi-words satoshi-words.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the executable
$ ./satoshi-words
^D��<GS>^A^DEThe Times 03/Jan/2009 Chancellor on brink of second bailout for banks
```

创建区块头

为了创建区块头，挖矿节点需要填充6个字段，见表8.3。

表8.3　区块头结构

大小	字段	描述
4 字节	版本（Version）	用于跟踪软件/协议更新的版本号
32 字节	前序区块哈希（Previous Block Hash）	链中前一个区块（父区块）的哈希值
32 字节	默克尔根（Merkle Root）	本区块交易默克尔树根的哈希
4 字节	时间戳（Timestamp）	区块大致创建时间（Unix 时间戳）
4 字节	难度目标（Difficulty Target）	本区块工作量证明算法的难度目标
4 字节	随机数（Nonce）	用于工作量证明算法的计数器

在区块277316被开采出来时，描述区块结构的版本号是2，以小字节序格式编码的4字节数字是0x02000000。

接着，挖矿节点需要添加“前序区块哈希”。即区块277315的区块头哈希，区块277315是景的节点从网络上接收到的最新区块，景已接受，并将其选定为候选区块277316的父区块。区块277315区块头的哈希是：
`0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569`

下一个步骤是将所有交易汇总成一棵默克尔树，以便计算并将默克尔根添加至区块头中。铸币交易将成为区块中的第一笔交易。然后，418笔其他交易添加在其后，最终总共有419笔交易被添加到区块中。正如我们在第7章“默克尔树”中看到的，树的叶子节点数量必须为偶数，所以需要将最后一笔交易复制一遍，形成420个叶子节点，每个节点均对应一笔交易的哈希值。交易哈希按对组合，继续进行哈希计算，从而

生成树的不同层次，直到所有交易被汇总到位于树“根”的节点。默克尔树的根将所有交易摘要汇总成一个32字节的数值，如例8-3所示的“默克尔根”。

```
c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e
```

接下来，挖矿节点添加上一个4字节的时间戳，以Unix“纪元（Epoch）”时间戳格式编码，它是以1970年1月1日零点（UTC/GMT时区）为起点，到目前经历的时间秒数的计时方式。时间1388185914与“2013年12月27日星期五23：11：54 UTC/GMT”对等。

再下一步，节点填充难度目标值，这个值定义了保证本区块有效的工作量证明难度的要求值。难度值在区块中以“难度位”度量标准进行存储，难度位是以“尾数-指数”格式编码的。这种编码格式含1字节的指数，紧跟3字节的尾数（系数）。举例来说，在区块277316中，难度位的值为0x1903a30c，第一部分0x19是十六进制的指数，第二部分0x03a30c为系数。难度目标的概念在“难度目标和目标调整”中有所描述，“难度位”的表示在本章“难度的表示法”中可以看到解释。

最后一个字段是随机数（nonce），初始化为0。

填充完所有字段后，区块头就完成了，而区块的挖矿过程也就可以开始进行了。现在的目标是找到一个随机数，使区块头的哈希小于难度目标。挖矿节点需要测试成千上万亿个随机数，直到找到一个满足要求的随机数值。

区块挖矿

现在候选区块已经被景的节点构建完成，是时候让硬件矿机来对这个区块进行“挖矿”了——找到工作量证明算法的解，使区块有效。在本书中，我们已经学习了加密哈希函数，它们在比特币系统的各个方面被广泛采用。SHA256是用于比特币挖矿过程的哈希函数。

简单地说，挖矿就是通过不断修改一个参数，重复计算区块头的哈希，直到找到一个与目标值匹配的哈希的过程。哈希函数的结果无法提前预知，也不能创建一个模式使其产生特定哈希。哈希函数的这个特性意味着，生成哈希结果并匹配特定目标的唯一途径就是不停地尝试，通过随机修改输入，生成不同哈希，直到碰巧得到希望的结果。

工作量证明算法

哈希算法利用任意长度的数据作为输入，生成一个固定长度的确定结果，即输入数据的数字指纹。对于任意特定的输入，结果总是相同的，只要实现了相同哈希算法，都可以轻易计算并验证。加密哈希算法的关键特性是对于两个不同的输入，几乎不可能生成相同的指纹。作为推论，给定一个数字指纹，除了不断尝试各种输入，没有其他办法可以构造一个数据，使其哈希值与给定指纹相同。

若采用SHA256算法，不管输入的长度是多少，其输出总是256位。在例8-8中，我们利用Python解释器来计算短语“I am Satoshi Nakamoto”（我是中本聪）的SHA256哈希。

例**8-8　SHA256**示例

```
$ python

Python 2.7.1
>>> import hashlib
>>> print hashlib.sha256("I am Satoshi Nakamoto").hexdigest()
5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e
```

例8-8显示了“I am Satoshi Nakamoto”的哈希值计算结果：5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e 这个256位的数字就是短语的哈希或者摘要，它依赖短语中的所有部分。如果增加一个字母、标点符号，或任何其他字符，都会导致不同哈希的生成。

现在，如果我们改变短语，将会得到一个完全不同的哈希。我们试着加一个数字到短语的末尾，仍然使用简单的Python脚本进行计算，如例8-9所示。

例**8-9　SHA256**，使用脚本通过迭代一个随机数产生多个哈希值

```
# example of iterating a nonce in a hashing algorithm's input

import hashlib

text = "I am Satoshi Nakamoto"

# iterate nonce from 0 to 19
for nonce in xrange(20):

    # add the nonce to the end of the text
    input = text + str(nonce)

    # calculate the SHA-256 hash of the input (text+nonce)
    hash = hashlib.sha256(input).hexdigest()

    # show the input and hash result
    print input, '=>',  hash
```

运行这个脚本将产生20个短语的哈希值，这些短语通过在文本最后添加一个数字而有所不同。通过增加数字，我们能够得到不同的哈希，如例8-10所示。

例**8-10　SHA256**的输出，使用脚本通过迭代一个随机数产生多个哈希值

```
$ python hash_example.py

I am Satoshi Nakamoto0 => a80a81401765c8eddee25df36728d732...
I am Satoshi Nakamoto1 => f7bc9a6304a4647bb41241a677b5345f...
I am Satoshi Nakamoto2 => ea758a8134b115298a1583ffb80ae629...
I am Satoshi Nakamoto3 => bfa9779618ff072c903d773de30c99bd...
I am Satoshi Nakamoto4 => bce8564de9a83c18c31944a66bde992f...
I am Satoshi Nakamoto5 => eb362c3cf3479be0a97a20163589038e...
I am Satoshi Nakamoto6 => 4a2fd48e3be420d0d28e202360cfbaba...
I am Satoshi Nakamoto7 => 790b5a1349a5f2b909bf74d0d166b17a...
I am Satoshi Nakamoto8 => 702c45e5b15aa54b625d68dd947f1597...
I am Satoshi Nakamoto9 => 7007cf7dd40f5e933cd89fff5b791ff0...
I am Satoshi Nakamoto10 => c2f38c81992f4614206a21537bd634a...
I am Satoshi Nakamoto11 => 7045da6ed8a914690f087690e1e8d66...
I am Satoshi Nakamoto12 => 60f01db30c1a0d4cbce2b4b22e88b9b...
I am Satoshi Nakamoto13 => 0ebc56d59a34f5082aaef3d66b37a66...
I am Satoshi Nakamoto14 => 27ead1ca85da66981fd9da01a8c6816...
I am Satoshi Nakamoto15 => 394809fb809c5f83ce97ab554a2812c...
I am Satoshi Nakamoto16 => 8fa4992219df33f50834465d3047429...
I am Satoshi Nakamoto17 => dca9b8b4f8d8e1521fa4eaa46f4f0cd...
I am Satoshi Nakamoto18 => 9989a401b2a3a318b01e9ca9a22b0f3...
I am Satoshi Nakamoto19 => cda56022ecb5b67b2bc93a2d764e75f...
```

每个短语均产生一个完全不同的输出。它们看起来完全随机，但是你可以在任何计算机上使用Python重新生成完全相同的结果，看到完全一样的哈希值。

在这类场景中，作为变量使用的数字叫作随机数。这个随机数用于

改变加密函数的输出，在本例中，它用于改变短语的SHA256指纹。

为了使这个算法具有挑战性，我们任意设置一个目标：找到一个短语，它的十六进制哈希值开始于0。很幸运，这个并不难！例8-10显示，短语“I am Satoshi Nakamoto13”的哈希值“0ebc56d59a34f5082aaef3d66b37a661696c2b618e62432727216ba953104
合我们的要求。经过13次尝试我们达成了目标，从概率的角度看，如果哈希函数的输出是均匀分布的，我们可以期望，每经过16次哈希计算就可以找到一个以十六进制0开头的结果（十六进制0到F的十六分之一）。用数字的角度看，就是找到一个比
0x1000000000000000000000000000000000000000000000000000000000000000
小的哈希值。我们把这个阈值叫作目标值（target），我们需要做的就是找到这样一个哈希值，使其数值小于这个目标值。如果缩小目标，要查找小于它的哈希值的任务，将变得更加困难。

如果做一个简单的类比，我们可以想象一个游戏，游戏玩家重复投一对骰子，试图找到一个小于特定目标的点数。在第一回合，目标是12，只要投的不是两个6，都会赢。第二轮，目标为11，玩家必须投出10或以下的点数才能赢，这轮仍然很简单。几轮过后，目标降到了5。现在，半数以上的投掷点数之和都会超过5，也就是无效的。随着目标值越小，有效投掷次数将呈指数级增加。最终，当目标降到2时（最小可能点数），赢的概率只剩下1/36，或者2%。

在例8-10中，获胜的“随机数”是13，这个结果可以被任何人独立确认。任何人都可以将13添加到短语“I am Satoshi Nakamoto”之后并计算哈希，验证结果是否小于目标值。成功结果也是工作量的证明，因为它能够证明我们已经做了足够多的工作并找到了随机数。虽然只要进行一次哈希计算就能进行验证，但是找到一个可用的随机数却需要进行13次的哈希计算。如果我们的目标值更小（难度更高），就需要更多次数的哈希计算，才能找到合适的随机数，但是任何人想验证这个结果，仍然

只需要进行一次哈希计算。此外，知道目标值后，任何人都可以利用统计学原理对计算难度进行估算，进而知道需要完成多少工作才能找到一个合适的随机数。

比特币的工作量证明与例8-10面临的挑战非常类似。首先，矿工创建一个填满交易的候选区块。接着，矿工计算区块头的哈希，看其是否小于当前的目标值。如果哈希不小于目标值，矿工就修改随机数（通常就是对随机数加1）并重新计算。在比特币网络当前的难度值下，矿工平均需要尝试千万亿（10^{15}）次以上，才能找到一个随机数，使得区块头的哈希值足够小。

例8-11是一个高度简化的工作量证明算法，基于Python实现。

例**8-11**　简化的工作量证明实现

```
#!/usr/bin/env python
# example of proof-of-work algorithm

import hashlib
import time

max_nonce = 2 ** 32 # 4 billion

def proof_of_work(header, difficulty_bits):

    # calculate the difficulty target
    target = 2 ** (256-difficulty_bits)

    for nonce in xrange(max_nonce):
        hash_result = hashlib.sha256(str(header)+str(nonce)).hexdigest()

        # check if this is a valid result, below the target
```

```
        if long(hash_result, 16) < target:
            print "Success with nonce %d" % nonce
            print "Hash is %s" % hash_result
            return (hash_result,nonce)

    print "Failed after %d (max_nonce) tries" % nonce
    return nonce


if __name__ == '__main__':

    nonce = 0
    hash_result = ''

    # difficulty from 0 to 31 bits
    for difficulty_bits in xrange(32):

        difficulty = 2 ** difficulty_bits
        print "Difficulty: %ld (%d bits)" % (difficulty, difficulty_bits)

        print "Starting search..."

        # checkpoint the current time
        start_time = time.time()
        # make a new block which includes the hash from the previous block
        # we fake a block of transactions - just a string
        new_block = 'test block with transactions' + hash_result

        # find a valid nonce for the new block
        (hash_result, nonce) = proof_of_work(new_block, difficulty_bits)

        # checkpoint how long it took to find a result
        end_time = time.time()

        elapsed_time = end_time - start_time
        print "Elapsed Time: %.4f seconds" % elapsed_time

        if elapsed_time > 0:

            # estimate the hashes per second
            hash_power = float(long(nonce)/elapsed_time)
            print "Hashing Power: %ld hashes per second" % hash_power
```

运行这段代码，你可以设置希望的难度（比特位，即头部多少位为0），看看需要多长时间才能找到一个解。例8-12显示的是在一台普通笔记本电脑上的工作情况。

例**8-12**　不同难度目标下运行工作量证明算法的结果

```
$ python proof-of-work-example.py*

Difficulty: 1 (0 bits)

[...]

Difficulty: 8 (3 bits)
Starting search...
Success with nonce 9
Hash is 1c1c105e65b47142f028a8f93ddf3dabb9260491bc64474738133ce5256cb3c1
Elapsed Time: 0.0004 seconds
Hashing Power: 25065 hashes per second
Difficulty: 16 (4 bits)
Starting search...
Success with nonce 25
Hash is 0f7becfd3bcd1a82e06663c97176add89e7cae0268de46f94e7e11bc3863e148
Elapsed Time: 0.0005 seconds
Hashing Power: 52507 hashes per second
Difficulty: 32 (5 bits)
Starting search...
Success with nonce 36
Hash is 029ae6e5004302a120630adcbb808452346ab1cf0b94c5189ba8bac1d47e7903
Elapsed Time: 0.0006 seconds
```

```
Hashing Power: 58164 hashes per second

[...]

Difficulty: 4194304 (22 bits)
Starting search...
Success with nonce 1759164
Hash is 0000008bb8f0e731f0496b8e530da984e85fb3cd2bd81882fe8ba3610b6cefc3
Elapsed Time: 13.3201 seconds
Hashing Power: 132068 hashes per second
Difficulty: 8388608 (23 bits)
Starting search...
Success with nonce 14214729
Hash is 000001408cf12dbd20fcba6372a223e098d58786c6ff93488a9f74f5df4df0a3
Elapsed Time: 110.1507 seconds
Hashing Power: 129048 hashes per second
Difficulty: 16777216 (24 bits)
Starting search...
Success with nonce 24586379
Hash is 0000002c3d6b370fccd699708d1b7cb4a94388595171366b944d68b2acce8b95
Elapsed Time: 195.2991 seconds
Hashing Power: 125890 hashes per second

[...]

Difficulty: 67108864 (26 bits)
Starting search...
Success with nonce 84561291
Hash is 0000001f0ea21e676b6dde5ad429b9d131a9f2b000802ab2f169cbca22b1e21a
Elapsed Time: 665.0949 seconds
Hashing Power: 127141 hashes per second
```

正如你所看到的，难度每增加1位，寻找解所需的时间呈指数增长。考虑整个256字节的数字空间，每次你将0的位数增加1个，就将搜索空间缩减了一半。在例8-12中，为找到前面26位为0的哈希，需要进行8400万次哈希计算才能找到合适的随机数。即使哈希速度超过每秒12万次，在一台普通笔记本电脑上也需要耗费10分钟才能找到解决方案。

在撰写本书时，比特币网络挖矿的要求是找到的区块头哈希值必须

小于
000000000000004c296e6376db3a241271f43fd3f5de7ba18986e517a243baa7。如你所见，这个目标哈希值开头有很多0，也就是说，可接受的哈希范围小了很多，因此，找到一个有效哈希也要困难得多。为了发现一个新的区块，全网平均每秒要进行1.5×10^{17}次哈希运算。看起来像是不可能完成的任务，但幸运的是，比特币网络目前已经具备了每秒10^{17}次哈希计算（100PH/sec）的处理能力，平均10分钟就可以找到一个新的区块。

难度的表示法

在例8-3中，我们看到区块中包含有难度目标，被称为“难度位（difficulty bits）”或者“位”，在区块277316中，它的值为0x1903a30c。这个标识以系数/指数的格式来表示难度目标，前2个十六进制数字代表指数，后面的6个十六进制数字是系数。在这个区块中，指数是0x19，系数是0x03a30c。

根据这个表示法计算难度目标的公式如下。

```
target = coefficient * 2^(8 * (exponent - 3))
```

套用公式，难度位的值为0x1903a30c，我们可以得到如下。

```
target = 0x03a30c * 2^(0x08 * (0x19 - 0x03))^

=> target = 0x03a30c * 2^(0x08 * 0x16)^

=> target = 0x03a30c * 2^0xB0^
```

用十进制表示如下。

```
=> target = 238,348 * 2^176^

=> target =
22,829,202,948,393,929,850,749,706,076,701,368,331,072,452,018,388,575,715,328
```

转换为十六进制如下。

```
=> target = 0x0000000000000003A30C0000000000000000000000000000000000000000
```

这意味着，一个有效的区块277316，其区块头哈希必须小于这个目标。按照二进制数字的写法，其前60位必须为0。基于这个级别的难度值，如果一个矿工每秒可以处理1万亿次哈希计算（1 tera-hash/秒或者1TH/sec），那么平均每经过8496个区块，他将有可能找到一个合适的区块头哈希，换句话说，平均59天就可能会找到一个新区块。

难度目标和目标调整

我们看到，目标确定了寻找区块的难度，进而影响了解决工作量证明算法所需的时间。那么问题来了：为什么难度是可调整的？谁来调整？如何调整？

比特币区块平均每10分钟生成一个，这是比特币的心跳机制，也是货币发行频率和交易处理速度的基础。保持这个恒定的速率不仅是短期目标，也需要长期维持。随着时间推移，计算机的处理能力会持续地快速提升。另外，参与挖矿的人数，以及他们使用的计算机也会不断变化。为维持10分钟的区块创建速度，挖矿的难度必须应这些变化进行调整。实际上，难度是个动态参数，它会周期性地调整，以适应10分钟挖出一个区块的目标。简而言之，难度目标设置为不管挖矿能力如何，新区块产生间隔时间都是10分钟左右。

那么，在一个完全去中心化的网络中是如何调整难度的呢？难度目标的调整是自动发生的，并且是在每个完全节点独立完成的。每经过2016个区块，所有节点都会调整工作量证明的难度。难度调整等式会测

算最后2016个区块的产生时间，并与预期时间20160分钟（2周时间，基于每个区块10分钟计算）进行比较。计算得出实际时间与预期时间的比值后，对难度进行相应调整（调高或调低）。简单来说，如果网络找到区块的时间快于10分钟，难度就会调高。如果区块发现时间慢于10分钟，则难度调低。

等式可归纳如下。

```
New Difficulty = Old Difficulty * (Actual Time of Last 2016 Blocks / 20160 minutes)
```

例8-13显示了在比特币核心客户端中使用的代码。

例**8-13**　工作量证明难度调整**——GetNextWorkRequired**（）**pow.cpp** 第**43**行

```
// Go back by what we want to be 14 days worth of blocks
const CBlockIndex* pindexFirst = pindexLast;
for (int i = 0; pindexFirst && i < Params().Interval()-1; i++)
    pindexFirst = pindexFirst->pprev;
assert(pindexFirst);
// Limit adjustment step
int64_t nActualTimespan = pindexLast->GetBlockTime() - pindexFirst->GetBlockTime();
LogPrintf("  nActualTimespan = %d  before bounds\n", nActualTimespan);
if (nActualTimespan < Params().TargetTimespan()/4)
    nActualTimespan = Params().TargetTimespan()/4;
if (nActualTimespan > Params().TargetTimespan()*4)
    nActualTimespan = Params().TargetTimespan()*4;

// Retarget
uint256 bnNew;
uint256 bnOld;
bnNew.SetCompact(pindexLast->nBits);
bnOld = bnNew;
bnNew *= nActualTimespan;
bnNew /= Params().TargetTimespan();

if (bnNew > Params().ProofOfWorkLimit())
    bnNew = Params().ProofOfWorkLimit();
```

参数Interval（2016区块）、TargetTimespan（2周或1209600秒）

在**chainparams.cpp**中定义。

为防止难度调整速度过快，每轮调整的幅度必须小于一个因子（4）。如果计算得出的难度调整需要超过因子4，将只调整到最大值4，而不是更大。额外的调整推迟到下个调整周期完成，因为这种不平衡状态会延续到下个2016区块。因此，一旦出现哈希算力大幅度变化，与难度形成巨大差异时，可能需要经过几个2016区块调整周期才能达到平衡。

找到一个比特币区块的难度，大约需要全网10分钟的处理世界，每完成2016个区块，就会基于最近2016个区块的寻找时间，重新调整一次。

需要注意的是，难度与交易数量或者交易价值无关。也就是说，维护比特币安全的哈希算力的容量及相应的电力消耗，也与交易数量完全无关。比特币可以横向扩展，获得更大范围的应用，即使哈希算力仍然维持当前的水平，其安全性也不会变化。哈希算力的提高，代表着市场力量驱使更多的矿工进入这个市场参与竞争并获取报酬。只要有足够的哈希算力控制在诚实挖矿、追逐奖励的矿工手里，就可以防止“接管”攻击，保证比特币足够安全。

难度目标与电力成本、比特币与支付电费的货币的交换汇率紧密相关。高性能采矿系统就是利用当前技术，制造高性能的计算设备，使其尽可能高效地将电能转换为哈希计算的能力。对挖矿市场最主要的影响因素是以比特币计价的每度电的价格，因为它决定了挖矿的收益，进而影响了人们对进入或退出挖矿市场的选择。

成功挖到区块

前面我们看到，景的节点创建了一个候选区块，并准备对其进行挖矿。景有几台带有专用ASIC芯片的硬件挖矿设备，芯片中无数的集成电路以不可思议的速度并行运行着SHA256算法。这些专用设备通过USB与挖矿节点相连。接下来，在景的桌面电脑上运行的挖矿节点会将区块头信息发送到挖矿硬件，这些硬件设备则以每秒千万亿次的速度测试不同随机数。

大约在开始挖矿11分钟后，某个挖矿设备找到了一个解，并将其回传给挖矿节点。将随机数4215469401填入区块头后，产生了如下的区块头哈希。

```
0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

这个哈希值小于当前目标。

```
0000000000000003A30C00000000000000000000000000000000000000000000
```

景的挖矿节点立即将区块传送给它的所有对等节点。这些节点接收并验证后，再次将区块向网络传播。当区块呈波纹状传遍网络时，每个接收到区块的节点都会将其加入自己的区块链本地副本中，将区块链扩展到包含277316个区块的新高度。当挖矿节点接收并验证这个区块后，它们将自己寻找相同高度区块的工作抛开，并立即进入下一区块的计算工作中。

在下一节，我们将研究每个节点验证区块和最长链选择的过程，这个过程将建立共识，并最终形成去中心化的区块链。

验证新区块

比特币共识机制的第三个步骤是网络上的每个节点独立验证新区块。当新发现的区块在网络中传播时，每个节点在将其继续发送到它的对等节点前，会进行一系列的测试工作，以验证其有效性。结果就是，只有有效的区块才会被传播到网络当中。独立验证也保证了诚实矿工挖出的新区块能被区块链接纳，并赢得奖励。而那些不诚实的矿工，其区块将被拒绝，不仅失去奖励，也浪费了寻找工作量证明解的努力，甚至连电力成本都无法得到补偿。

当一个节点接收到新区块时，它将依据一个长长的规则列表对其进行验证，如果不符合任一要求，区块将被拒绝。这些规则可以在比特币核心客户端的函数CheckBlock和CheckBlockHeader中查到，主要包括如下内容。

●区块数据结构的语法正确。

●区块头哈希比目标难度小（确保满足工作量证明要求）。

●区块的时间戳早于未来2小时（允许时间错误）。

●区块大小在允许范围内。

●第一个交易（只有第一个）是一个铸币交易。

●区块中的所有交易有效，通过交易检查列表验证（参看本章中“独立交易验证”）。

每个新区块均接受所有节点的独立验证，这样确保了矿工不能进行欺诈。在前面几节中，我们研究了矿工们如何创建一笔特殊交易，以获

得在此区块中产生的新比特币和交易费。为什么矿工不能随意创建一个交易给自己发送1000比特币，而只能获得正确的奖励金额呢？原因在于每个节点都是基于相同的规则来验证区块的。一个无效的铸币交易会导致整个区块无效，并被其他节点拒绝，永远无法成为账本的一部分。矿工必须创建完美的区块，基于所有节点接受的公共规则，并且根据正确的工作量证明方法进行挖矿。为了完成这个证明，他们已经投入大量的电力挖矿，如果涉嫌欺诈，所有的电力投入和挖矿努力都将付之东流。这就是为什么独立验证会成为去中心化共识的重要组成部分。

组装和选择区块链

比特币共识机制的第四个步骤是区块链的组装和最大累积工作量（最大难度）的区块链的选择。一旦节点完成一个新区块的验证，它将尝试将区块和已存在的区块链进行连接，形成新的链条。

节点维护着三套区块的集合：连接到主区块链的区块；形成主链分支的区块（次链）；在已存在的链中均找不到父区块的新区块（孤儿）。只要不符合验证规则的任意一条，无效区块会立即被拒绝，不会被加入任何区块集合中。

“主链”在任何时候都是拥有最大累积难度的区块链。在大多数情况下，这也是含有区块最多的链，例外的情况是，同时存在两条长度一样的链，但其中一条的工作量证明更多。主链也可能存在分支，分支上有与主链区块存在同辈关系的区块。这些区块虽然同样是有效的，但不是主链的一部分。保留这些分支的目的在于，某些分支可能会在未来得以延伸，并在难度值上超过主链，那么后续的区块就会引用它们。在下一小节（“区块链分叉”）中，我们将看到由于几乎同时挖出了相同高度的区块，导致了次链的产生。

当节点接收到新区块时，会尝试将其插入已存在的区块链中。首先在区块中检查“前序区块哈希”字段，它是新区块对其父区块的引用。接着，在已存在的区块链中查找这个父区块。大多数时候，父区块会在主链的“顶端”，意味着新区块延长了主链。举例来说，新区块277316有一个到其父区块277315的哈希的引用。大多数节点接收到区块277316前，在其主链上已经存在区块277315，并处于区块链顶部，接收277316后将其与主链相连，并将区块链延长。

有时，正如“区块链分叉”中所描述的，新区块延长了一个链，但它

却不是主链。在这种情况下，节点将区块连接到次链上，然后比较次链和主链的累积难度。如果次链的累积难度超过了主链，该节点将在次链上重新收敛，也就是说它将选择次链作为新的主链，而原来的主链则成了次链。如果该节点是矿工，它之后将在这个更新更长的链上构建区块，并进一步延长它。

如果接收到的是一个有效的区块，但是其父区块却无法在已存在的链上找到，那么这个区块就被当作是一个“孤儿”区块。孤儿区块被保存到孤儿区块池中，直到它的父区块到达本节点。一旦接收到父区块并连接到已存在的链条上，孤儿区块就可以从孤儿区块池中被取出，并与其父区块相连，成为区块链的一部分。孤儿区块的产生通常是因为两个区块被挖出的时间间隔比较短，而接收顺序刚好相反（子区块先于父区块到达本节点）。

通过选择最大难度的链条，所有节点最终将达成全网范围内的共识。当更多的工作量证明加入，某个可能的链条延长时，不同链条间暂时性的差异，最终将会得到解决。挖矿节点根据它们的挖矿能力，通过创建新区块的方式，“投票”决定待延长的链条。当它们开采出新的区块并延长了区块链条时，这个新区块本身就代表了它们的投票结果。

在下一节中，我们将讨论如何通过独立选择最长难度链条来解决竞争链（分叉）间的差异问题。

区块链分叉

因为区块链是一个去中心化的数据结构，区块链的不同副本不是时刻都能保持完全一致的。区块在不同节点间的到达时间会存在差异，因而不同的节点可能拥有不完全一样的区块链视图。为了解决这个问题，每个节点总是选择并尝试延长代表了最大工作量证明的区块链，即最长

的链条或者最大累积难度的链条。通过汇总记录在链上的每个区块的难度值，节点可以计算出创建这个链所耗费的工作量证明总额。只要所有节点都选择最大累积难度的链条，全局比特币网络最终将调整到一致状态。分叉是不同版本的区块链间暂时的不一致现象，当更多区块被加入其中某一个分支时，最终收敛将解决这一问题。

在接下来的几张图中，我们将在全网范围内，追踪“分叉”事件的整个过程。图形是比特币作为一个全局网络的简化表示。在现实中，比特币网络拓扑并没有按照地理位置进行组织。相反，互连的节点形成了一个网状网络，但是相连的节点间地理距离可能非常遥远。使用地理拓扑的表示法是为了更加简洁地演示分叉。在真实的比特币网络中，节点间的“距离”是用节点到节点间的“跳数”（hops）来衡量的，而不是基于它们的物理位置。出于演示目的，不同区块用不同颜色表示，散布在网络中，而它们经过的连接也用该颜色标示出来。

第一个图例（见图8.2），全网拥有统一的区块链视图，蓝色的区块位于主链的顶端。

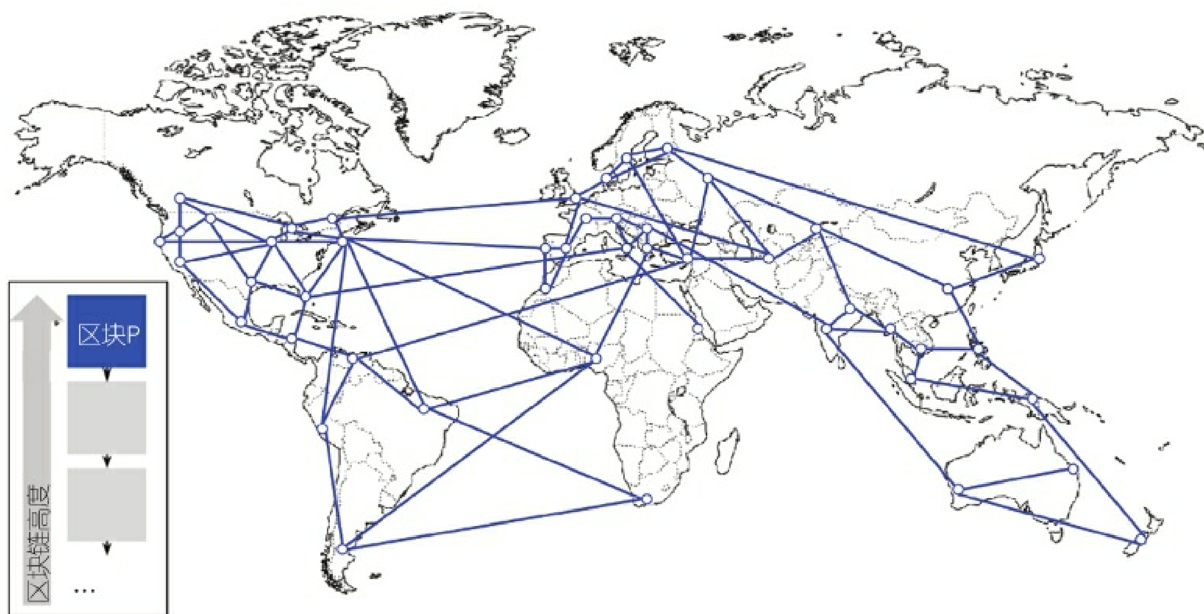


图8.2 区块链“分叉”事件演示——分叉前

当两个候选区块同时竞争以形成最长区块链时，“分叉”发生了。通常情况下，这是因为两个矿工几乎在同时找到了工作量证明算法的解。当矿工发现各自候选区块工作量证明的解时，他们立即将“赢得”的区块发送给离他们最近的邻居，这些邻居则继续将区块传播到网络中。每个节点接收到一个区块后，都会将其整合进自己的区块链，将区块链延长一个区块。如果节点稍后又看到另一个候选区块同样延伸了相同的父区块，那么它将会把第二个候选区块连接到次链上。结果是，有些节点先“看到”这个候选区块，而其他节点则会先看到另外一个候选区块，这样两个竞争版本的区块链就出现了。

图8.3中，我们看到两个矿工几乎同时发现了两个不同的区块。两个区块均是蓝色区块的子区块，也就是说，将在蓝色区块之上创建新区块来延伸区块链。为了便于跟踪，一个区块被标识为红色，从加拿大发起，另一个标为绿色，从澳大利亚发起。

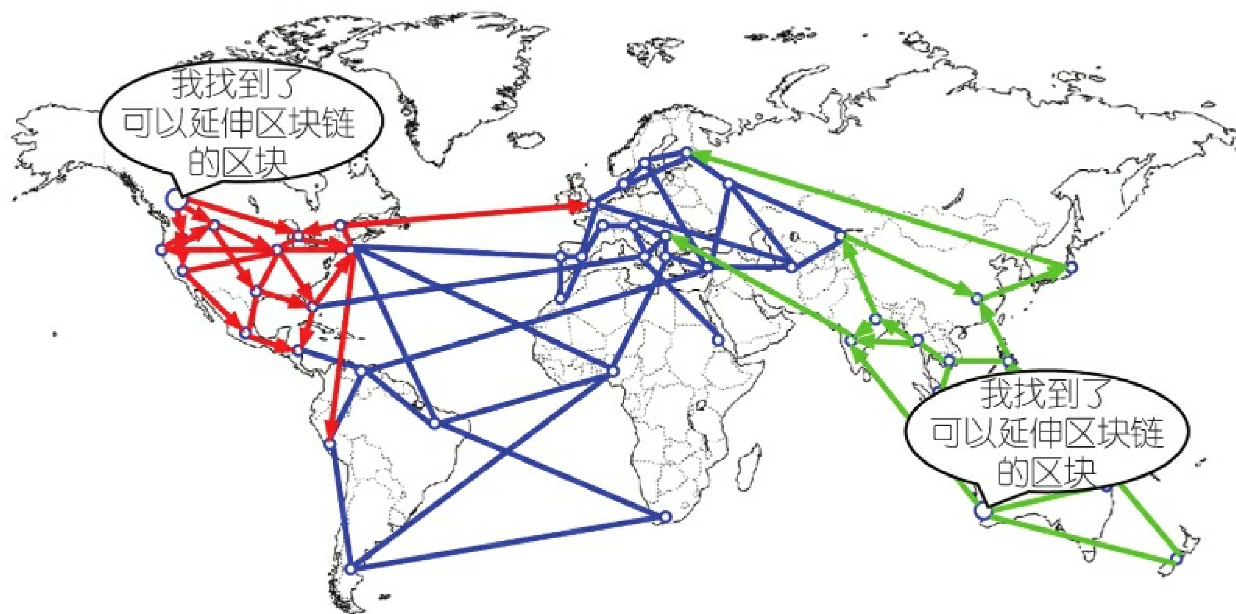


图8.3 区块链“分叉”事件演示：两个区块同时被发现

假设A国的矿工找到了“红色”区块的工作量证明的解，从而在父区块“蓝色”区块上延伸了区块链。几乎同时，B国的矿工也找到了“绿色”区块的解，也在“蓝色”区块上延伸了区块链。现在，存在两个可能

的区块，一个是“红色”，来自A国；另一个是“绿色”，来自B国。两个区块都是有效的，都包含了有效的工作量证明，都在相同的位置延伸了区块链。两个区块中的交易也几乎相同，可能只是在交易顺序上有些细微差异。

当两个区块都开始广播后，有些节点先接收到了“红色”区块，而有些节点则先收到了“绿色”区块。如图8.4所示，网络分裂为两个不同的区块链视图，一个视图顶端是“红色”区块，另一个则是“绿色”区块。

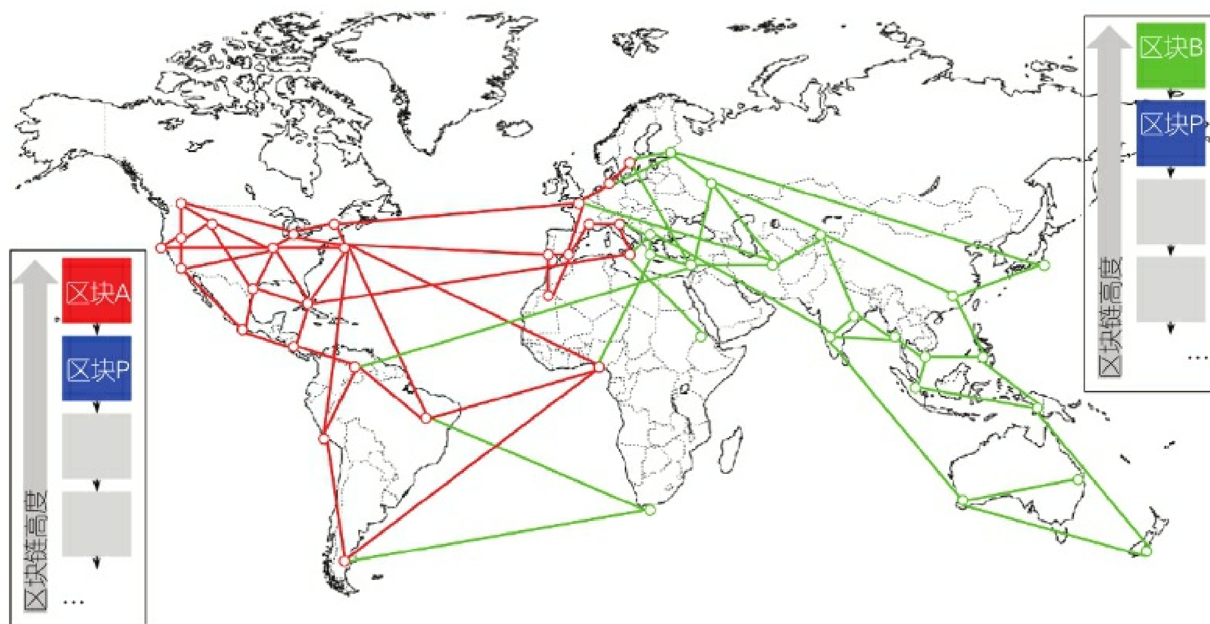


图8.4 区块链“分叉”事件演示：两个区块的广播导致网络的分裂

从这个时刻开始，离A国的挖矿节点最近的（拓扑上而不是地理上最近）比特币网络节点将首先接收到“红色”区块，并建立一个最大累计难度的区块链，“红色”区块为这个链的最后一个区块（蓝—红），同时忽略晚到的“绿色”候选区块。同时，接近B国的节点将认为“绿色”区块是赢家，并将其加入区块链的最后区块（蓝—绿），而忽略晚到的“红色”区块。任何先看到“红色”的矿工，会以“红色”为父区块，立即开始构建新的候选区块，并尝试找到候选区块的工作量证明解。先接受“绿色”区块的节点则基于区块链顶端的“绿色”区块开始工作。

分叉问题几乎都能在一个区块的处理过程中就得以解决。网络中的一部分哈希算力专注于在“红色”区块之上挖矿，而另一部分算力则专注在“绿色”区块之上挖矿。即便哈希算力能做到平均分配，其中某个阵营先找到新区块的工作量证明解，并广播到网络的概率也是极高的。我们假定在“绿色”区块上挖矿的阵营先找到一个“粉色”的区块，从而延长了区块链（蓝—绿—粉）。它们会立即将此区块广播到网络，整个网络都会确认其有效性。如图8.5所示。

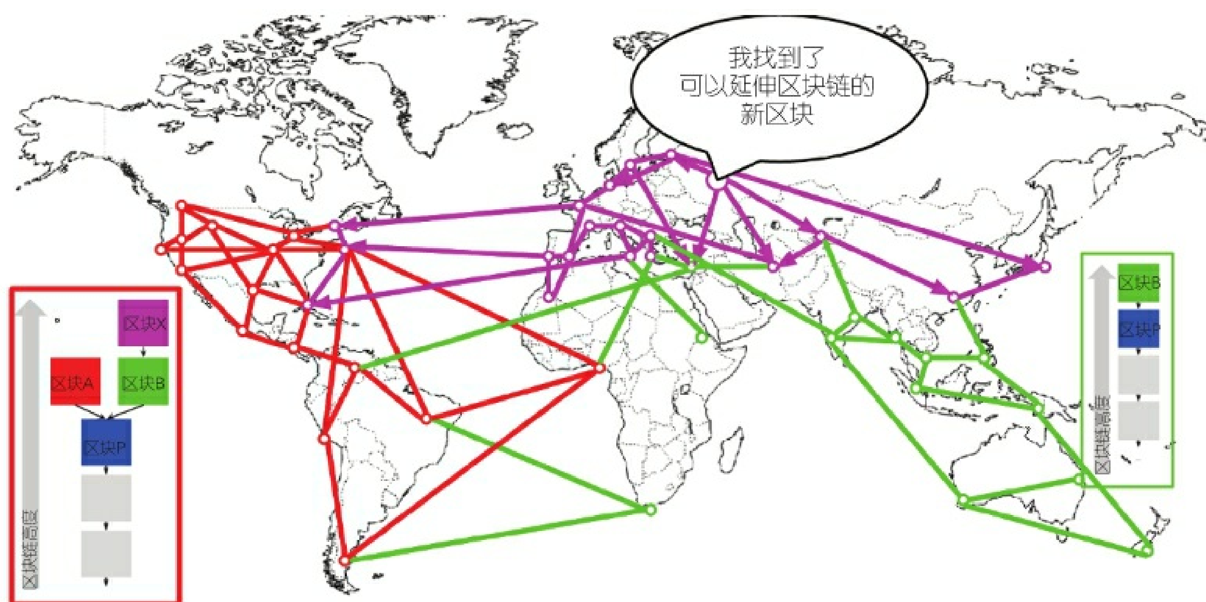


图8.5 区块链“分叉”事件演示：一个区块延伸了一个分叉

所有在上轮挖矿竞争中选择“绿色”并成为赢家的节点，简单地将区块链延伸一个区块。选择“红色”的，现在将看到两条链：“蓝—绿—粉”和“蓝—红”。“蓝—绿—粉”链相对“蓝—红”更长（累积难度更多）。结果，这些节点将重新将“蓝—绿—粉”设置为主链，而“蓝—红”则成为次链，如图8.6所示。这就是链的重收敛，因为那些节点被迫改变它们对区块链的认定，以接受更长链条的客观事实。所有正在尝试延长“蓝—红”链的节点将放弃在那条链上的工作，因为父区块“红色”已不在最长的链条上，它们的候选区块变成了“孤儿”。由于区块已不在最长的主链中，“红色”区块内的交易只能重新进入队列，等待处理下一个区块。

整个网络收敛到一个区块链，“蓝—绿—粉”，“粉色”则作为链条的最后一个区块。所有矿工立即投入新的、以“粉色”区块为父区块的延展“蓝—绿—粉”区块链的工作。



图8.6 区块链“分叉”事件演示：网络重新收敛到一个区块链

理论上，如果连续两个区块几乎同时被处于分叉两侧的矿工挖出，则存在分叉延续两个区块的可能性。但是发生这种事情的概率非常低。一个区块的分叉可能每个星期会发生一次，但是两个区块的分叉极少出现。

10分钟的比特币区块间隔是一种在快速确认（交易结算）和分叉可能性之间的一种权衡。更短的区块间隔时间可以使交易的清算更快，但是会导致更频繁的区块链分叉；而更长的区块间隔时间虽然可以降低分叉次数，却使得交易结算变慢了。

挖矿和哈希竞赛

比特币挖矿是一个高度竞争的行业。哈希算力自比特币诞生以来每年都呈指数级增长。一些年份的增长反映了彻底的技术更新，比如2010和2011年，很多矿工从CPU挖矿转到了GPU挖矿，以及现场可编程门阵列（FPGA）挖矿。在2013年，随着ASIC挖矿的引入，把SHA256函数直接集成到了挖矿的专用芯片上，导致了哈希算力的另一次巨大飞跃。第一台采用这种芯片的矿机所产生的算力，比2010年整个比特币网络的算力还要大。

比特币运行前五年全网哈希算力的情况如下所示。

2009年

0.5 MH/秒~8 MH/秒（16倍增长）。

2010年

8 MH/秒~116 GH/秒（14500倍增长）。

2011年

16 GH/秒~9 TMH/秒（562倍增长）。

2012年

9 TH/秒~23 TH/秒（2.5倍增长）。

2013年

23 TH/秒~10 PH/秒（450倍增长）。

2014年

10 TH/秒~150 PH/秒（截至8月份，15倍增长）。

图8.7显示了两年间比特币网络哈希算力的增长情况。如你所见，矿工之间的竞争以及比特币的成长导致了哈希算力（全网络每秒的哈希运算能力）呈指数级增长。

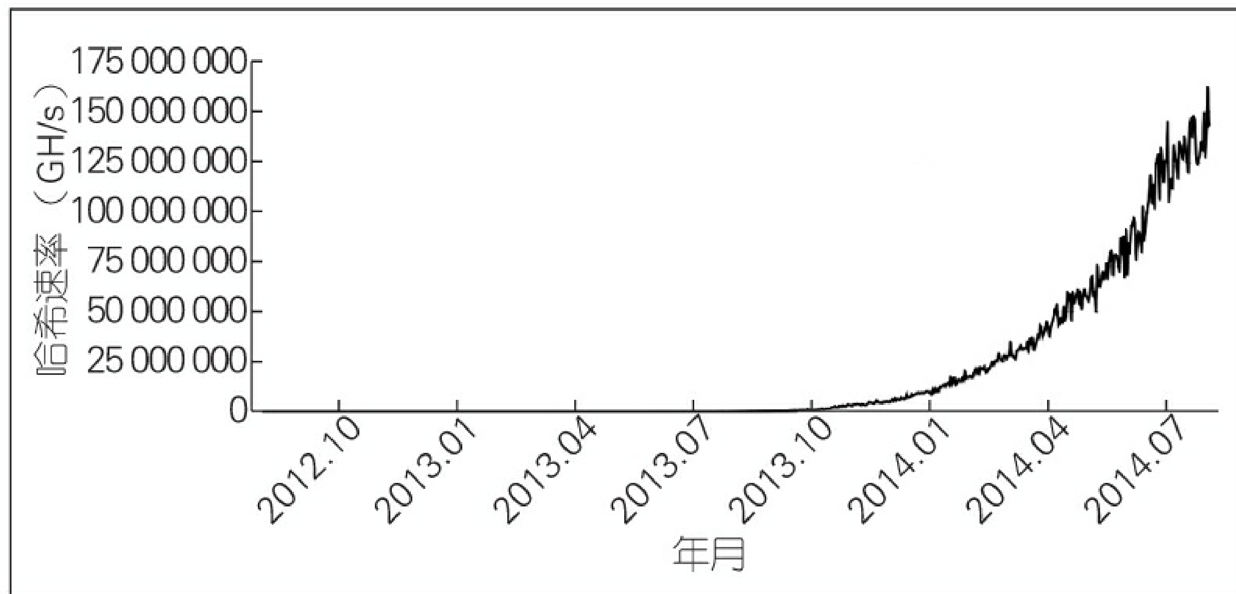


图8.7　两年间的总哈希算力变化

资料来源：blockchain.info。

随着投入挖矿运算的哈希算力呈爆炸式增长，挖矿难度也相应地提高了。图8.8中，难度值以当前难度与最小难度（第一个区块的难度）之间的比率来计量。

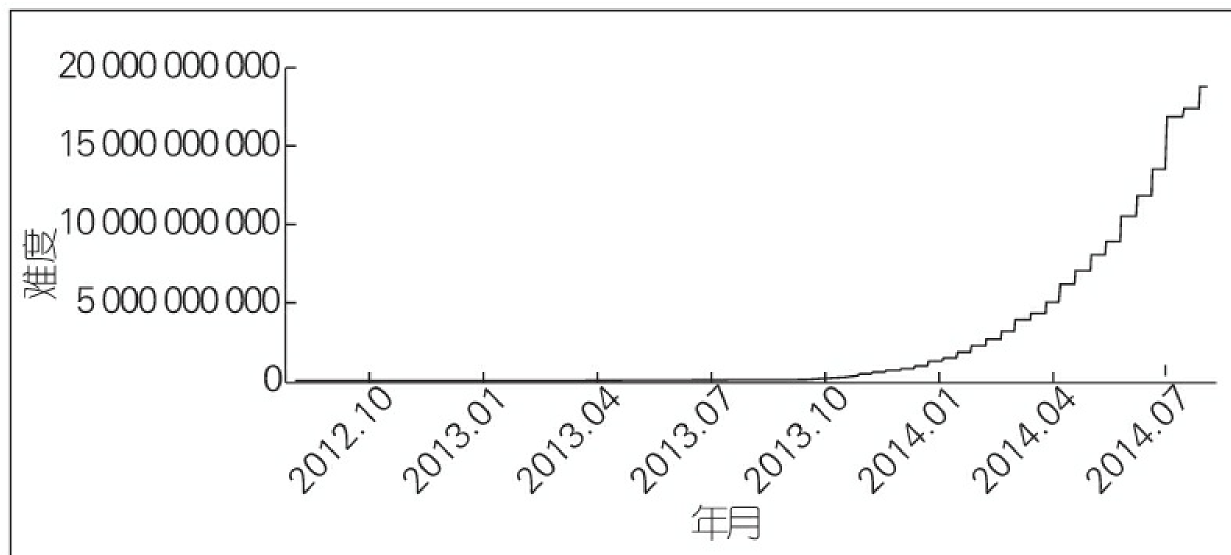


图8.8　两年间的比特币挖矿难度

资料来源：blockchain.info。

2012~2014年，ASIC挖矿芯片的集成度越来越高，已经接近了芯片制造业最前沿的22纳米特征尺寸（分辨率）的水平。由于挖矿利润的驱使，这个行业甚至比通用计算行业发展得更快。当前，ASIC制造商的目标是超过通用CPU芯片制造商，设计出16纳米特征尺寸的芯片。目前看来，比特币挖矿已经很难有巨大的飞跃。这个行业已经达到了摩尔定律的极限，摩尔定律认为每18个月计算密度翻一番。尽管如此，随着更高密度芯片的出现，以及能够部署成千上万芯片的更高密度数据中心的发展，网络算力还将继续保持指数级的增长。现在已经不再是比较单一芯片的能力，而是如何把更多芯片集成在一起，并处理好散热和供电的问题。

扩展随机数方案

从2012年起，比特币社区提出了一种区块头结构的基础限制问题的解决方案。在比特币发展的早期，矿工可以不断迭代随机数找到一个区

块，使其哈希值小于指定目标。随着难度的增加，矿工们经常碰到即使将40亿个随机数都循环一遍也找不到解的情况。不过，这个问题因为区块中时间戳的更新而轻易地得到了解决。由于时间戳是区块头的一部分，它的改变使得矿工可以重新进行随机数迭代，得出不同的结果。但是，当挖矿硬件性能超过4GH/秒时，这个方法就变得越来越难了，因为随机数的值1秒钟内就耗尽了。当ASIC挖矿装备加入后，运算能力超过了TH/秒级，挖矿软件需要更大的随机数空间来寻找有效的区块。时间戳虽然可以延后一点，但是延后太多又会导致区块无效。区块头需要一个能够产生“变化”的源。解决方案是使用铸币交易作为额外随机数的来源，因为币基脚本可以存储2～100字节的数据。矿工开始使用这个空间作为扩展的随机数空间，这使得他们可以探索更大范围的区块头以找到有效的区块。铸币交易受到默克尔树的保护，也就是说，币基脚本的任何修改都会导致默克尔根的变化。8字节的扩展随机数，加上4字节的“标准”随机数，允许矿工在不改变时间戳的情况下，每秒尝试2^{96}（8后跟28个0）种可能性。如果未来，矿工们有能力遍历所有这些可能性，他们仍然可以通过改变时间戳来进一步增加可能性。当然，币基脚本中也仍然有富余空间供未来扩展随机数使用。

矿池

在这种激烈竞争的环境下，单个矿工（也被称为“个体矿工”）独立工作基本没有机会赢得竞争。他们通过挖矿弥补电力和硬件成本的可能性非常低，基本上就等同于参加一场赌博或者买彩票。即使最快的消费型ASIC挖矿系统，也无法赶上那些安装了成千上万芯片的商业系统，那些系统通常建在水电站附近的巨大机房里。矿工们一般联合起来组成矿池，他们将算力集中起来，而奖励也在成员间共享。通过加入矿池，矿工可以得到全部奖励的一小部分，但是这种方式削减了不确定性，矿工们几乎每天都能分到奖励。

我们来看一个具体的例子。假定一个矿工购买了一台矿机，其处理能力能达到6000G哈希每秒（GH/s），或者6TH/s。在2014年的8月份，这台设备的成本大概是1万美元。设备运行时的功率为3kW，每天72度（kWH）耗电，每天平均电力成本是7～8美元。在当前的比特币挖矿难度下，如果矿工单干的话，大概每155天或者5个月可以找到1个新区块。如果矿工在这个时间内真的找到了1个区块，那么他将得到25比特币的奖励，每个比特币兑换价格大概是600美元，总的奖励是1.5万美元。这笔钱扣除硬件和电力的全部成本后，大概还有3000美元的收益。但是，5个月内能否找到一个区块，完全凭矿工的运气。他可能5个月内找到2个区块，获得更多的收益；也有可能10个月也找不到1个区块，导致财务损失。更糟糕的是，比特币工作量证明的难度很可能在这段时间内已经显著提高了，以当前哈希算力的增长速度，矿工最多在6个月时间内必须达到收支平衡，否则硬件就将过时，需要被更强的挖矿硬件取代。如果矿工加入一个矿池，每周就能赚到500～700美元，而不用等待5个月一次的“横财”。矿池定期派发的奖励帮他摊销了硬件和电力成本，不再需要承担巨大的风险。硬件设备仍然会在6～9个月后过时，风险依然很高，但是至少在这个时间内的回报是定期发放，相对可靠的。

矿池通过专门的矿池协议，将成百上千的矿工集合在一起。个体矿工在矿池中创建账号后，通过设置把挖矿设备与矿池服务器相连。在挖矿的过程中，挖矿硬件保持与矿池服务器相连，与其他矿工同时进行挖矿工作。这样，矿池中的矿工共享挖矿的努力，也共享收获的奖励。

挖矿成功后，奖励将被发送到矿池的比特币地址，而不是某个个体。当矿工的份额达到某个阈值后，矿池服务器将定期把奖励发送到矿工的比特币地址。通常，矿池服务器会抽取一定百分比的佣金，作为提供矿池服务的报酬。

寻找候选区块工作量证明解的工作被分割成多个部分，分给所有加入矿池的矿工，这些矿工则根据其贡献赚取奖金份额。矿池通常设置一

个比比特币网络的实际难度小1000倍的难度目标，用以衡量矿工的工作量投入，以分割奖励份额。当矿池中有人成功挖出一个区块，奖励由矿池领取，矿池再根据矿工贡献的大小分配奖金。

矿池对所有矿工开放，不管是大还是小，专业还是业余。因此，一个矿池中不仅存在只有单台小型矿机的矿工，也存在拥有大量高端挖矿硬件的矿工。有些矿工挖矿耗费功率只有几十千瓦，有些则运营着功耗达到兆瓦级的数据中心。矿池如何才能做到既可以避免欺诈，又能基于每个矿工的贡献公平分配奖金呢？答案在于使用比特币的工作量证明算法来衡量矿工的贡献，矿池将难度值设得很低，确保即使是最小的矿工也能经常赢得奖励份额，让他们觉得加入矿池是值得的。通过设置较低的分享份额的难度目标，矿池可以衡量每个矿工完成的工作量。每当矿工找到一个小于矿池难度的区块头哈希，就证明了他已完成了寻找结果的哈希计算。更重要的是，这些为获取份额而做的工作，能以一个可衡量的统计方法，为整个矿池寻找小于比特币网络难度目标的哈希做出贡献。成百上千的矿工寻找小值哈希，最终总能找到一个足够小的、满足比特币网络难度目标的区块哈希。

我们回到骰子游戏的例子。假设骰子玩家的目标是总点数小于4点（全网难度），矿池可以设置一个较为简单的目标，比如，计算每个参与矿池的玩家掷出小于8的总点数的次数。当矿池中的玩家投掷出小于8点（矿池份额目标）时，他们赢得一个份额，但他们并没有赢得游戏，因为还没有达到游戏的目标（小于4点）。矿池玩家可以比较容易地就达到矿池目标，从而非常有规律地赢得他们的份额，即使最终他们没有达到赢得游戏的目标。

时不时地，矿池中的玩家会投出一个总点数小于4点的组合，让矿池赢得游戏。接着，基于玩家们赢得的份额进行收益分配。虽然目标设置为8或更少并没有最终赢得游戏，但这是衡量玩家们投掷点数的公平方法，而且偶尔也会产生一个小于4的点数。

类似地，一个矿池可以设置矿池的难度，确保矿池中的矿工可以经常找到小于矿池难度的区块头哈希，从而赢得份额。这种尝试工作时常也会找到一个小于比特币网络目标的区块头哈希，从而产生有效区块，矿池成为这个区块的赢家。

托管矿池

大多数矿池都是“托管矿池”，即公司或者个人运营着矿池服务器。矿池服务器的拥有者被称为矿池经营者，他按一定比例向加入矿池的矿工抽取奖励费用的佣金。

矿池服务器运行着专用的软件，根据矿池挖矿协议来协调矿工们的工作。矿池服务器同时与一个或多个完全比特币节点相连，可以直接访问区块链数据库的完整复制。这使得矿池服务器可以代表矿工对区块和交易进行验证，使他们从运行完全节点的负担中解脱出来。对于矿池中的矿工而言，这是一个重要的考虑因素，因为一个完全节点需要一台专用的计算机，配置至少15G~20G的硬盘，至少2G的内存（RAM）。此外，运行在完全节点上的比特币软件还需要时不时地监控、维护、升级。任何因为缺少维护或者资源而导致的宕机，都会损害矿工的收益。对于很多矿工来说，不用维护完全节点即能参与挖矿是加入托管矿池的另一大好处。

矿池矿工利用挖矿协议，比如Stratum（STM）或GetBlockTemplate（GBT）与矿池服务器相连。曾经有个旧标准，被称为GetWork（GWK），到2012年年底基本上已经作废了，因为这个标准难以支撑哈希速度超过4GH/s的挖矿工作。不管是STM还是GBT协议，都会创建一个包含候选区块头的区块模板（**templates**）。矿池服务器通过归集交易，添加铸币交易（包括扩展随机数空间），计算默克尔根，加入连接到前序区块的哈希引用，从而创建一个新的区块。候选区块的区块头作为模板，发送给矿池中的所有矿工。每个矿工基于区块模板，在一个低于比特币网络难度值的目标下进行挖矿；一旦找到满足矿池难

度值的区块，就将其发回矿池服务器，赢得奖励份额。

P2P矿池

托管矿池有可能引发矿池管理员的欺诈行为，他可能将矿池的算力引导至双重支付交易或者无效区块（参见本章中“共识攻击”）。此外，中心化的矿池服务器也存在单点故障的隐患。如果矿池服务器宕机或者因拒绝服务攻击而放慢运行速度，矿池中的矿工就无法挖矿。在2011年，为解决这个中心化问题，引进了一个新的矿池挖矿协议：P2P矿池（P2Pool），它是一个点对点的矿池，不需要中心管理员。

P2P矿池通过将矿池服务器的功能去中心化，实现了一个平行的类似区块链的系统，叫作份额链（**share chain**）。份额链是一条相比比特币区块链具有较低难度的区块链。份额链允许矿工在去中心化的矿池中协同工作，它们以每30秒一个份额区块的速度在份额链上进行份额挖矿。份额链上的每个区块记录着参与贡献的矿工的奖励份额，并且继承了之前份额区块上的份额记录。当某个份额区块同时达到比特币网络的难度目标时，这个区块就会被传播出去，并入比特币区块链，而区块奖励则根据每个矿工之前对份额的贡献度进行发放。本质上，份额链采用一种类似比特币区块链的去中心化共识机制，让所有矿池中的矿工都能跟踪所有贡献份额的记录，而不像矿池服务器一样，由一个中心节点保存矿工的份额和奖励记录。

P2P矿池挖矿比托管矿池挖矿复杂得多，它要求矿工运行一台具有足够硬盘空间、内存和网络带宽的专用电脑，以支持完全节点和P2P矿池节点软件。P2P矿池矿工将他们的挖矿设备与本地的P2P矿池节点进行连接，P2P矿池节点通过向挖矿设备发送区块模板的方式，模拟矿池服务器的功能。在P2P矿池中，矿工构建他们自己的候选区块，归集交易，其行为与个体矿工很类似，但是P2P矿工是在份额链上合作挖矿。P2P矿池整合了个体矿工和矿池两者的优势，能够让个体矿工在支出上具有更加细粒度的优势，在控制层面，它不需要像托管矿池那样将控制

权交给矿池管理人。

当前，矿池中集中的算力已接近可发起51%攻击的能力（参见本章中“共识攻击”），出于对此的担忧，P2P矿池的矿工数量有了显著增长。P2P矿池协议进一步的发展有望移除对完全节点的依赖，从而使这种去中心化的挖矿更加易用。

共识攻击

如果矿工（或矿池）想要利用自身拥有的哈希算力进行欺骗或攻击的话，比特币的共识机制至少在理论上是有可能被攻击的。正如我们所看到的，共识机制依赖大多数矿工出于个人利益而愿意诚实行事的假设前提。但是，如果一个矿工或矿工集团能获得全网较大比例的挖矿能力时，他们就可以通过攻击共识机制从而瓦解比特币网络的安全性和可用性。

值得注意的是，共识攻击只会影响将来的共识，或者最多也只能影响到“不久前”（过去的几十个区块）的时段。比特币的账本随着时间推移将越来越稳定。虽然在理论上允许任何深度的分叉存在，但在实践中，要迫使非常深的分叉产生需要极其巨大的算力，因此老的区块实际上是不可变的。共识攻击也不会威胁私钥和签名算法（ECDSA）的安全。共识攻击无法盗取比特币，无法不带签名地使用比特币，也不能转移比特币，不能改变过去的交易或所有权记录。共识攻击只能影响最近的区块，并且通过拒绝服务攻击来破坏将来的区块生成。

一种针对共识机制的攻击叫作“51%攻击”。这是由一群控制了51%以上的全网哈希算力的矿工，合谋发起的对比特币的攻击。他们拥有开采大部分区块的能力，可以故意在区块链中制造“分叉”、进行双重支付交易，或者针对特定交易或地址发起拒绝服务攻击。分叉/双重支付攻击指的是攻击者通过在某个区块之下创建分叉，形成新的主链，使之前已确认的区块变得无效。只要拥有足够的算力，攻击者就可以将连续6个甚至更多区块变为无效，从而使那些经过6次确认、已被认定为是无法篡改的交易变为无效状态。另外，需要注意的是，双重支付只能应用于攻击者自己创建的交易，因为他们只能对这些交易进行有效签名。如果通过使交易失效，攻击者可以得到不可逆的交换品，或者可以购买产

品而不用付钱，那么对自己创建的交易进行双重支付是有利可图的。

我们来看一个“51%攻击”的实际例子。在第一章中，我们观察过爱丽丝和鲍勃之间购买一杯咖啡的交易。在不等爱丽丝的付款交易获得确认（区块挖矿）的情况下，咖啡店老板鲍勃自愿将咖啡给爱丽丝，因为这样可以为顾客提供相对快速便捷的服务，一杯咖啡出现双重支付的风险实在很小。这与咖啡店接受25美元以下没签名的信用卡支付的风险差不多，因为通常信用卡拒付的风险很低，而为了等待顾客签名而使交易延迟的成本相对却大得多。相比之下，使用比特币购买昂贵商品而出现双重支付的风险就高多了，买家可以通过传播一个和真实交易UTXO一样的交易，从而取消发给商户的支付交易。双重支付可以有两种方式：要么在交易确认之前；要么攻击者利用区块链分叉使多个区块无效。“51%攻击”允许攻击者在新链上进行双重支付，从而使旧链上相应的交易失效。

在我们的例子中，恶意攻击者马洛里（Mallory）到卡罗尔的画廊购买了一幅漂亮的将中本聪描绘成普罗米修斯的三联画。卡罗尔以相当于25万美元的比特币价格，将这幅叫作“大火”的画卖给了马洛里。卡罗尔并没有等待6次甚至更多确认，而是只等了1次确认就把画打包好交给了马洛里。保罗（Paul）是马洛里的同谋，经营着一个巨大的矿池，在马洛里的交易被打包进一个区块后，保罗马上发起了一次“51%攻击”。他指挥其矿池对包含刚才那笔交易的区块进行了重新计算，将区块中马洛里支付给卡罗尔的交易替换成了一笔双重支付的交易。这笔双重支付使用了相同的UTXO，并把收款人地址改为马洛里钱包的地址，也就是说，比特币又回到了马洛里手里。接着保罗继续指挥其矿池在新的区块上进行计算，挖出新的区块，使得包含双重支付的区块链比原来那条链更长（即在包含马洛里购买画作交易的区块之下产生分叉）。当新的区块链分叉高度高于原来的链后，双重支付交易取代了原来的真实交易。卡罗尔不仅失去了画作，也没有收到比特币支付款。在整个过程中，保罗矿池里的矿工可能都无法察觉到“双重支付”的存在，因为挖矿程序是

自动运行的，无法对每个区块或交易进行监控。

为防止此类攻击，商户出售高额产品时，至少应该等待6次以上确认才将货品交给顾客。或者，商户可以使用第三方多签名账户，同样等待多次确认，直到第三方账户资金到账后才将商品交给顾客。确认次数越多，“51%攻击”越难以对交易进行篡改。对于高额商品，通过比特币支付依然是便利高效的，即使买家必须等待24小时（保证得到144次确认）才能发货。

除了双重支付攻击，还有一种共识攻击是针对特定的比特币参与者（特定比特币地址）的拒绝服务。拥有大多数算力的攻击者可以简单地将特定交易忽略掉。如果这些交易被其他矿工包含进区块，攻击者可以对该高度的区块排除该笔交易后进行重新计算，形成分叉。只要攻击者控制了网络中的大部分算力，这种类型的攻击就可以持续地对特定地址或地址集进行拒绝服务。

尽管名为“51%攻击”，这种攻击并不需要真正掌握51%以上的算力。实际上，更低比例的哈希算力就可以尝试发起“51%攻击”。51%的阈值意味着一旦达到这个水平，攻击几乎肯定能成功。共识攻击本质上是对下一区块的争夺，“强壮”的一方就更容易成功。哈希算力越小，攻击成功的概率就越低，因为其他拥有“诚实”挖矿能力的矿工控制着区块的生成。从另一个角度看，攻击者拥有越多的哈希算力，他所创建的分叉就越长，也就有可能篡改越久远的区块，或者控制更多将来产生的区块。一些安全研究小组已经使用统计模型证明了存在几种共识攻击，只要拥有30%的哈希算力就能成功。

全网哈希算力的大量增加，已使得比特币系统不可能被单个矿工攻击了。个体矿工几乎不可能控制即使1%的全网算力。但是，矿池带来的中心化控制，也导致了矿池管理人以盈利为目的攻击风险。托管矿池中的矿池管理人控制了候选区块的生成，因而也控制了交易的筛选。这给了矿池管理人排除或包含交易的权利。如果这种权利被矿池管理人有

节制而巧妙地进行滥用，矿池管理人就可以在不为人知的情况下发起共识攻击并获益。

但是，不是所有的攻击都是因利益驱使的。一种潜在的攻击情形就是攻击者只是想捣毁比特币网络，而不希望从中获益。这些意在破坏比特币网络的恶意攻击者，需要有大笔的资金和隐蔽的计划，若是由资金充足的政府支持，则一切都顺理成章。或者，一个资金充足的攻击者也可以购置大量挖矿设备并联合一些矿池管理人，对其他矿池发起拒绝服务攻击，从而达到对比特币的共识机制发动攻击的目的。这些情形都在理论上存在可能性，随着比特币网络的全网算力持续呈指数级增长，要发起这些攻击已经不切实际了。比特币系统在不断演化，诸如旨在推动矿池去中心化控制的P2P矿池协议的推出，比特币的共识机制已变得越来越难以攻击。

毋庸置疑，严重的共识攻击会在短期内侵蚀对比特币的信任，并导致其价格下跌。但是，比特币网络和软件均在持续进化，共识攻击也会面临比特币社区的及时应对，比特币网络一定会比过去任何时刻都更坚实，更稳健，更强大。

第9章 替代链、替代币、应用程序

比特币是分布式系统和货币研究20年的成果，它带来了一场新技术的革命：基于工作量证明的去中心化共识机制。这项比特币的核心发明已引发了一系列领域的技术革新浪潮，包括货币、金融服务、经济、分布式系统、投票系统、公司治理、合约等。

在本章中，我们将观察一些比特币及区块链发明的分支：替代链、替代币，以及自2009年这项新技术引入以来开发的应用程序。当然，我们主要还是研究替代币（alternative coins，简称**alt coins**）[注]，这些是基于比特币的设计模式创建的数字货币系统，运行于完全独立的区块链和网络。

替代币如此多，我们难免挂一漏万，这可能会让这些替代币的创建者或者粉丝感到愤怒。但是本章的目的并不是评估或评价替代币，甚至也不想基于某些主观因素而特意提及自认为最重要的替代币。实际上，我们突出介绍的例子主要是基于每种创新的第一个应用，或具有显著差异的代表性币种，以此来展示生态系统的广泛性和多样性。从货币的角度看，某些最有意思的替代币实际上是完全失败的。但这反而使对它们的研究变得更有意思，同时也强调了一个事实，本章不能作为替代币的投资指南。

每天都有新的替代币诞生，要避免遗漏某些重要的替代币几乎是不可能的，而这些被遗漏的替代币也许将会改变历史。这个领域的创新速度如此让人兴奋，我敢保证在本书出版时，本章已经变得不完整并且可能早就过时了。

---

1. 国内也有称之为“山寨币”的，为避免贬低嫌疑，书中统一译为“替代币”。——译者注

替代币与替代链的分类

比特币是个开源项目，它的代码是很多其他软件项目的基础。从其代码衍生出的软件项目的最常见形式就是各种去中心化货币或者叫替代币，它们均基于相同的基础构建区块以实现数字货币。

有一系列应用是在比特币区块链之上实现的协议层。包括元币（**meta coins**）、元链（**meta chains**）、区块链应用（**blockchain apps**），它们要么把区块链作为基础平台，要么新增协议层对比特币协议进行扩展。案例包括彩色币（Colored Coins）、万事达币（Mastercoin）、未来币（NXT）、合约币（Counterparty）等。

在下节中，我们将讨论几种比较著名的替代币，比如莱特币（Litecoin）、狗币（Dogecoin）、弗雷币（Freicoin）、素数币（Primecoin）、点点币（Peercoin）、暗黑币（Darkcoin）、零币（Zerocoin）等。这些替代币之所以著名，不是因为它们是最值钱或者是“最好”的替代币，而是因为某些历史环境，或者因为它们是特定类型替代币的最佳案例。

除了替代币，还有一系列不同的区块链实现，它们不是真正的“币”，所以我们把它们称为替代链（**alt chains**）。这些替代链实现了共识算法和分布式账本技术，可以作为合约、名称注册或其他应用的基础平台。替代链使用相同的基础技术构建区块，有时也使用货币或令牌作为其支付手段，但是它们的主要目的不是充当货币。作为替代链的例子，我们也将对域名币（Namecoin），以太坊（Ethereum）和未来币等进行研究。

除了比特币中使用的工作量证明共识算法，还存在其他一些实验性质的共识协议，如基于资源证明、发布证明等的共识协议。我们将以

Maidsafe和Twister为案例了解此类共识机制。

最后，还有数量不少的比特币竞争者，他们提供数字货币或数字支付网络，但是没有使用去中心化的账本或基于工作量证明的共识机制，比如瑞波网络（Ripple）等。这些非区块链技术已超出本书范围，我们在本章中将不会讨论。

元币平台

元币和元链是比特币之上的软件层实现，它们要么实现了“币中币”功能，要么在比特币系统中叠加了一层平台/协议。这些功能层扩展了比特币核心协议，通过将额外数据进行编码、使其进入比特币交易或比特币地址的方式，扩展了比特币的特性及能力。元币的第一个实现使用了一系列技巧，将元数据加入比特币区块链中，包括使用比特币地址来编码数据，或者将新协议层的元数据编码进入交易的保留字段（比如，交易序号字段）。自从OP_RETURN交易脚本操作符引入后，元币可以更直接地将元数据添加到区块链中，因此大多元币已迁移到这种脚本中。

彩色币

彩色币是一种元协议，它利用了小额比特币存储信息。“彩色”币是用以指代另一种资产的一定金额的比特币。打个比方，在一张1美元的纸钞上盖个戳，戳上写着，“这是Acme公司1股股票的持股凭证”。现在这种1美元纸钞代表了两重含义：它不仅是一张钞票，也是一张持股凭证。作为持股凭证，其价值更高，你肯定不愿意拿这张钞票去买个糖果。因此，这张纸钞实际上已经不会再作为货币来使用了。彩色币工作原理与这个例子一样，将一个特定的很小金额的比特币转变为另外一种资产的交易证明。所谓“彩色”是指通过添加诸如颜色属性的标识来赋予比特币特殊含义的做法，它只是一种隐喻，并不是实指真正的颜色关联，彩色币上也不会真的有彩色。

彩色币由特定的钱包软件管理，这些钱包软件负责记录和解释附着在彩色币上的元数据信息。通过使用这种钱包软件，用户可以将一定金

额的比特币通过添加特定含义的标签转换成彩色币。标签可以代表股权证明、优惠券、真实的财产、商品或者可收集的令牌。如何对“颜色”含义进行赋值及解释，完全由彩色币用户决定。为了对比特币进行“染色”，用户需要先定义相关的元数据，比如发行类型、是否可以拆分、符号、描述，以及其他相关信息。一旦染色完成，这些比特币就可以进行买卖、拆分、聚合、接收股息等。彩色币也可以进行“褪色”，只要将特殊的关联信息去除，即可恢复为其面值代表的比特币。

为了演示彩色币的使用，我们创建20个“MasterBTC”彩色币，这些彩色币代表免费获取本书复制的优惠券，如例9-1所示。这些MasterBTC彩色币的每个单元都可以向任何使用兼容彩色币钱包的比特币用户出售和赠予，而获得彩色币的用户可以继续转让，或使用彩色币向发行方索取本书的免费复制。以下是代码（**http://cpr.sm/FoykwrH6UY**）。

例**9-1**　彩色币的元数据配置文件，记录免费获取本书复制的优惠券

```
{
  "source_addresses": [
    "3NpZmvSPLmN2cVFw1pY7gxEAVPCVfnWfVD"
  ],
  "contract_url": "https://www.coinprism.info/asset/
3NpZmvSPLmN2cVFw1pY7gxEAVPCVfnWfVD",
  "name_short": "MasterBTC",
  "name": "Free copy of \"Mastering Bitcoin\"",
  "issuer": "Andreas M. Antonopoulos",
  "description": "This token is redeemable for a free copy of the book \"Mastering
Bitcoin\"",
  "description_mime": "text/x-markdown; charset=UTF-8",
  "type": "Other",
  "divisibility": 0,
  "link_to_website": false,
  "icon_url": null,
  "image_url": null,
  "version": "1.0"
}
```

万事达币

万事达币（Mastercoin）是建立在比特币之上的一个协议层，它为那些多重扩展了比特币系统功能的应用提供基础平台。万事达币使用MST作为其货币代号，用于万事达币交易，但是万事达币的主要功能并不是货币本身。实际上，它是一个构建诸如用户货币、智能资产证明、去中心化资产交易、合约等应用的平台。可以将万事达币想象为建立在比特币金融交易传输层上的应用层协议，就像运行在TCP之上的HTTP协议。

万事达币的交易主要通过一个叫作“exodus”（出埃及记）的地址（1EXoDusjGwvnjZUyKkxZ4UHEf77z6A5S4P）进行，就像HTTP总是使用特定的TCP端口（80端口）来区分它的流量和其他TCP流量一样。万事达币协议现在正逐渐从使用特殊的exodus地址及多重签名技术迁移到使用OP_RETURN操作符来编码交易元数据。

合约币

合约币（Counterparty）是另一种建立在比特币之上的协议层实现。合约币让用户可以自定义货币、可交易令牌、金融工具、去中心化资产交易，以及其他特性。合约币主要使用OP_RETURN来记录元数据，这些元数据通过添加额外含义，增强了比特币交易的功能。合约币使用货币符号XCP来执行合约币交易。

替代币

大部分替代币从比特币的源代码衍生而来，也被称为“分叉（forks）”。有些则仅仅基于区块链模型，不使用比特币源代码，堪称“白手起家”。替代币和替代链（下节讨论）均为独立的区块链技术实现，使用其自有的区块链。两者的区别在于，替代币主要用于货币，而替代链主要不是货币应用，而是为其他目的服务。

严格地说，第一个主要的比特币“替代者”不是替代币，而是一条替代链，叫域名币，我们将在下节中进行讨论。

按照发布时间的先后顺序，首个替代币是比特币的一个分叉，出现于2011年8月，被称作**IXCoin**。IXCoin修改了比特币的一些参数，特别是将奖励金额增加到每区块96币，从而加快了货币的创建速度。

2011年9月，**Tenebrix**诞生。Tenebrix是第一个使用工作量证明替代算法的加密货币，其算法叫作**scrypt**，最早是用于密码增强的（防止暴力破解）。Tenebrix声称其目标是采用内存密集型算法，防止使用GPU和ASIC进行挖矿。Tenebrix作为货币来说并不成功，但是它是莱特币的基础，而莱特币取得了巨大成功，并由此衍生出了几百个分叉版本。

莱特币，不仅使用scrypt作为工作量证明算法，还将区块生成速度调快，相对比特币的10分钟1个区块，莱特币仅需要2.5分钟。莱特币被吹捧为“银子”（比特币则是“金子”），其目标是建立一套轻量级的替代货币。它更快的确认速度以及8400万的货币总量限制，使许多莱特币的追随者相信它比比特币更适合零售交易。

替代币在2011和2012年持续增长，这些替代币要么基于比特币，要么基于莱特币。到2013年初，总共有20种替代币在争夺市场地位。2013

年底，这个数字已暴增到200种，因此2013年也被称为“替代币之年”。2014年，替代币的增长还在继续。在撰写本书时，已有超过500种替代币，一半以上的替代币分叉来自莱特币。

创建一种新的替代币非常简单，这也就是为什么会有500多种替代币的原因。大多数替代币与比特币大同小异，没有什么研究价值。实际上，大多数只是创建者谋利的工具而已。但是，在充斥着抄袭和炒作圈钱的替代币市场中，也有一些例外，有些甚至是重要的创新。这些替代币要么采用截然不同的实现方法，要么在比特币的设计模式中添加了重要创新。这些替代币与比特币的差别主要体现在如下3个方面。

●不同的货币策略。

●不同的工作量证明算法或共识机制。

●特定的功能，比如更强的匿名手段。

要获取更多信息，可以查看网页**http://mapofcoins.com**（替代币和替代链的时间线示意图）。

评估替代币

市场上存在着如此多的替代币，如何判断哪个替代币更值得关注？有些替代币旨在成为货币被广为使用，有些则是具有不同特性及货币模型的实验产品，还有很多则是创建者快速致富的工具。为了评估一个替代币，我的做法是观察它们的典型特性以及市场表现。

这里的几个问题可以判断一个替代币相比比特币到底好在哪里。

●替代币有没有引入重大的创新？

●不同点是否具有足够的吸引力，能将用户从比特币吸引过来？

●替代币是否解决了某些有意思的利基市场或者实际应用问题？

●替代币能否吸引足够的矿工，以抵抗共识攻击并保证系统安全？

以下是几个需要考虑的财务和市场指标问题。

●替代币的市场容量如何？

●预估有多少该替代币的用户/钱包软件？

●有多少商户愿意接受这种替代币？

●每日交易量有多少？

●每日交易金额是多少？

在本章中，我们主要关注上述的第一个问题集，即替代币的技术特性和创新潜力。

货币属性不同的替代币：莱特币、狗币、弗雷币

比特币的一些货币属性使其成为一种独特的、发行量固定的通货紧缩货币。它的发行总量限定为2100万个主货币单位（或者2.1×10^{15}更小货币单位），发行速率呈几何递减；另外，它有10分钟的区块“心跳”，用于控制交易确认速度和货币产生速度。很多替代币修改了主要参数，以达到实现不同货币策略的目的。在几百个替代币中，最值得关注的包括以下几个。

莱特币

莱特币（Litecoin）是最早的替代币之一，发布于2011年，是比特币之后最成功的数字货币系统。它最主要的创新在于采用了scrypt作为工作量证明的替代算法（继承于Tenebrix），以及它更快/更轻量的货币参数。

●区块创建时间：2.5分钟。

●货币总量：到2140年达到8400万。

●共识算法：Scrypt工作量证明。

●市场容量：2014年年中达到1.6亿美元。

狗币

狗币（Dogecoin）创立于2013年12月，是莱特币的一个分叉。狗币之所以引人关注，主要是因为其货币发行速度快和货币总量大，其目的是鼓励用户消费和支付小费。狗币另一个引人关注之处是它起源于一个玩笑，却很快流行起来，拥有一个巨大而活跃的社区，不过它从2014年开始很快就衰败了。

●区块创建时间：60秒。

●货币总量：2015年达到100000000000（1000亿）。

●共识算法：Scrypt工作量证明。

●市场容量：2014年年中达到1200万美元。

弗雷币

弗雷币（Freicoin）于2012年7月创立。它是一种滞期费货币（**demurrage currency**），也就是说，它是负利率的。存储在弗雷币系

统中的货币价值每年将被收取4.5%的费用，从而促使用户尽量消费，不鼓励囤积货币。弗雷币引人注意的地方是其货币政策刚好与比特币的通货紧缩策略相反。弗雷币作为一种货币并不成功，但却是替代币中体现货币多样性的一个有趣例子。

●区块创建时间：10分钟。

●货币总量：到2140年达到10亿。

●共识算法：SHA256工作量证明。

●市场容量：2014年年中达到13万美元。

共识创新：点点币、多彩币、黑币、维理币、未来币

比特币的共识机制是基于SHA256算法的工作量证明。第一款引入scrypt作为工作量证明替代算法的替代币，提供了一种主要依赖CPU的挖矿方法，避免了因过度使用ASIC芯片而带来的算力集中问题。从那以后，共识机制的创新以惊人的速度持续发展。替代币纷纷采用不同的算法来实现共识机制，比如：scrypt、scrypt-N、Skein、Groestl、SHA3、X11、Blake等。有些替代币则综合多种算法来实现工作量证明。2013年，我们看到了代替工作量证明的发明——权益证明（**Proof of Stake**），它奠定了很多新型替代币的基础。

权益证明是一种货币拥有者可以将自己的货币作为有息抵押品的系统。有点类似存款凭证（CD），参与者可以预留其所持货币的一部分，同时以新货币（系统以利息支付的方式发行新货币）和交易费用的形式获取投资回报。

点点币

点点币（Peercoin）于2012年8月引入，是第一种混合工作量证明和权益证明算法发行新货币的替代币。

●区块创建时间：10分钟。

●货币总量：无限。

●共识算法：（混合）权益证明、工作量证明。

●市场容量：2014年年中达到1400万美元。

多彩币

多彩币（Myriad）发布于2014年2月，它同时使用了5种不同的工作量证明算法（SHA256d、Scrypt、Qubit、Skein、Myriad-Groestl），每种算法的难度变化取决于参与矿工的情况。目的是使系统免受ASIC专业化和集中化的影响，防止共识攻击，因为攻击者必须同时发起对多种挖矿算法的攻击。

●区块创建时间：平均30秒（每种挖矿算法2.5分钟）。

●货币总量：到2024年达到20亿。

●共识算法：多种工作量证明算法。

●市场容量：2014年年中达到12万美元。

黑币

黑币（Blackcoin）发布于2014年2月，使用权益证明算法。值得注意的是，它引入了“多池”的概念，一种可以在不同替代币间基于收益率自动切换的矿池。

●区块创建时间：1分钟。

●货币总量：无限。

●共识算法：权益证明。

●市场容量：2014年年中达到370万美元。

维理币

维理币（VeriCoin）是在2014年5月创建的，它使用权益证明共识算法，并采用随市场供需关系动态变化的利率机制。它也是首款可以在钱包软件中自动兑换成比特币进行支付的替代币。

●区块创建时间：1分钟。

●货币总量：无限。

●共识算法：权益证明。

●市场容量：2014年中达到110万美元。

未来币

未来币（NXT，读音同“Next”），是一种“纯粹”的权益证明替代币，它完全不使用工作量证明进行挖矿。未来币是一个全新的加密货币实现，没有使用比特币或其他替代币的代码。未来币实现了很多高级特性，包括名称注册（类似域名币）、去中心化的资产交易（类似彩色币）、集成的去中心化的加密消息功能（类似比特信，Bitmessage）、权益委托（将权益证明委托给别人）。未来币的拥戴者称其为“下一代”或2.0版的加密货币。

●区块创建时间：1分钟。

●货币总量：无限。

●共识算法：权益证明。

●市场容量：2014年年中达到3000万美元。

双目标挖矿创新：素数币、治疗币、格雷德币

比特币的工作量证明算法只有一个目的：维护比特币的网络安全。相对于传统支付系统的安全性，挖矿的成本并不是特别高，但它还是被很多人批评为“浪费”。下一代的替代币试图解决大家的这个关注点。双目标工作量证明算法在生成工作量证明、保护网络安全的同时，解决了挖矿的“有用性”问题。将外部使用添加到货币安全性中的做法带来了新的风险，即供需曲线受到外部影响。

素数币

素数币（Primecoin）发布于2013年7月，它的工作量证明算法是搜索素数，计算坎宁安（Cunningham）和bi-twin素数链。素数在很多科学领域都很有用。素数币的区块链中包含了已发现的素数，由此素数币不仅保存了交易的公共账本，还创建了一个公共的科学发现记录。

●区块创建时间：1分钟。

●货币总量：无限。

●共识算法：基于素数链发现的工作量证明。

●市场容量：2014年年中达到130万美元。

治疗币

治疗币（Curecoin）发布于2013年5月，它结合了SHA256工作量证

明算法和基于Folding@Home项目的蛋白质折叠研究。蛋白质折叠是一个需要进行大量计算的蛋白质生化反应模拟，用于发现治疗疾病的新药物。

●区块创建时间：10分钟。

●货币总量：无限。

●共识算法：工作量证明及蛋白质折叠研究。

●市场容量：2014年年中达到5.8万美元。

格雷德币

格雷德币（Gridcoin）于2013年10月发布。它结合了基于scrypt的工作量证明算法以及参与BOINC开放网格计算的贡献。伯克利开放式网络计算平台（Berkeley Open Infrastructure for Network Computing，简称BOINC）是一项用于科学研究网格计算的开放协议，允许参与者将他们富余的计算周期贡献出来，用于大范围的学术研究计算。格雷德币将BOINC作为一个通用的计算平台，而不是用于解决特定科学问题，比如素数寻找或者蛋白质折叠。

●区块创建时间：150秒。

●货币总量：无限。

●共识算法：工作量证明及BOINC网格计算贡献。

●市场容量：2014年年中达到12.2万美元。

关注匿名性的替代币：零币、CryptoNote、百特币、门罗币、暗黑币

比特币常被误认为是一种“匿名”货币。实际上，在比特币中，将用户身份与比特币地址联系起来还是相对简单的。利用大数据分析手段，将地址与使用者相关联，可以形成某人的消费习惯图谱。有几种替代币试图通过加强匿名性来解决这个问题。第一个进行尝试的很可能是零币，它是一种元币协议，用于在比特币之上提供匿名性。零币的概念是在2013年《IEEE安全与隐私论文集》中的一篇论文中被提出的。在撰写本书时，一种基于零币概念的全新替代币——零币正在开发当中。另外一种增强匿名性的尝试源于2013年10月发表的一篇论文，这种替代币叫作**CryptoNote**。CryptoNote应用了一种基础技术，已经有不少替代币的分叉实现了该技术，我们稍后将进行讨论。除了零币和CryptoNote，还有其他一些独立的匿名币，比如暗黑币，它们使用隐秘地址或者采用交易混杂的技术来提供匿名性。

零币

零币（Zerocoin/Zerocash）是一种数字货币匿名性的理论实践，于2013年由约翰·霍普金斯大学（Johns Hopkins University）的研究人员提出。Zerocash是基于零币理论实现的一种替代币，目前还处于开发中，尚未发布。

CryptoNote

CryptoNote是一个替代币的参考实现，它提供了一种匿名数字货币的基础，于2013年10月被引入。CryptoNote的设计目标是供各种不同程序复用其代码，它本身内置了定期重置机制，无法作为货币使用。已经有几个基于CryptoNote实现的替代币，包括百特币（Bytecoin，BCN）、Aeon（AEON）、Boolberry（BBR）、duckNote（DUCK）、Fantomcoin（FCN）、门罗币（Monero，XMR）、MonetaVerde（MCN）、Quazarcoin（QCN）等。值得注意的是，CryptoNote是一种全新构建的加密货币，不是比特币的分叉[注]。

百特币

百特币（Bytecoin）是基于CryptoNote技术的第一个实现，提供了一个可行的匿名货币系统。百特币出现于2012年7月。需要注意的是，之前也出现过一个叫“百特币”的数字货币，不过它的符号是BTE，而从CryptoNote发展而来的百特币货币符号是BCN。百特币使用Cryptonight工作量证明算法，这种算法要求每个实例至少占用2M内存，这使其无法采用GPU或ASIC进行挖矿。百特币继承了CryptoNote的环形签名、不可链接的交易，以及区块链防分析的匿名机制。

●区块生成时间：2分钟。

●货币总量：1840亿BCN。

●共识算法：Cryptonight工作量证明。

●市场容量：2014年年中达到300万美元。

门罗币

门罗币（Monero）是CryptoNote的另一个应用。它的发行曲线比百特币稍微平缓，将在头4年发行80%的货币。它同样继承了CryptoNote提供的匿名特性。

●区块生成时间：1分钟。

●货币总量：1840万XMR。

●共识算法：Cryptonight工作量证明。

●市场容量：2014年年中达到500万美元。

暗黑币

暗黑币（Darkcoin）于2014年1月发布。通过一个名为DarkSend的混淆协议来实现匿名货币。暗黑币与众不同的地方在于，它采用了11轮不同的哈希算法来完成工作量证明，这些算法包括：blake、bmw、groestl、jh、keccak、skein、luffa、cubehash、shavite、simd、echo。

●区块生成时间：2.5分钟。

●货币总量：最高2.2亿DRK。

●共识算法：多算法、多轮次工作量证明。

●市场容量：2014年年中达到1900万美元。

---

1. 由于一些替代币没有约定俗成的中文名称，本书中对这类名称不进行翻译。——译者注

非货币替代链

替代链是区块链设计模式的替代实现，主要用途不是货币。虽然大多替代链也包含一种货币，但货币主要作为分配其他东西的一个令牌，比如分配资源、合约等。换句话说，货币不是平台的关键特性，最多只能算是次要特性。

域名币

域名币（Namecoin）是基于比特币代码的第一个分叉。它是一个使用区块链的去中心化的“键—值”注册和传输平台。它支持全局域名注册，与互联网上的域名注册系统类似。域名币现在用于根域名.bit的替代域名服务（DNS）。域名币也可以用于在其他命名空间中注册名称及“键—值”配对；或者用于记录类似email地址、加密密钥、SSL证书、文件签名、投票系统、股权证书等；也用于很多其他应用程序。

域名币系统包含域名货币（货币符号NMC），用于支付注册和传输域名的交易费用。当前的价格下，注册一个域名的费用大概是0.01NMC，约合1美分。类似比特币系统，费用由域名币的矿工收集。

域名币的基本参数与比特币一样。

●区块生成时间：10分钟。

●货币总量：到2140年达到2100万NMC。

●共识算法：SHA256工作量证明。

●市场容量：2014年年中达到1000万美元。

域名币的命名空间没有限制，任何人都能以任何形式使用任何命名空间。但是，特定的命名空间有特定的规范。只有这样，当从区块链中读取时，应用软件才知道如何进行解析。如果命名空间不符合规范，不管用什么软件来解析都会产生错误。以下是比较常见的几个命名空间。

●d/域名命名空间，用于.bit域名。

●id/用于存储个人身份信息，比如email地址，PGP秘钥等。

●u/是一个附加的、更加结构化的个人信息存储规范（基于openspecs）。

域名币客户端与比特币核心客户端很类似，都是从相同的源代码演化而来的。安装完成后，客户端会下载一份完整的域名币区块链，接着就可以对名称进行查询和注册了。包括3个主要的命令。

```
name_new
```

查询或预注册一个域名。

```
name_firstupdate
```

注册一个域名并将其公开。

```
name_update
```

修改细节或刷新域名注册。

举例来说，为了注册一个mastering-bitcoin.bit域名，我们使用命令如下。

```
$ namecoind name_new d/mastering-bitcoin

[
    "21cbab5b1241c6d1a6ad70a2416b3124eb883ac38e423e5ff591d1968eb6664a",
    "a05555e0fc56c023"
]
```

name_new命令通过产生一串域名哈希和一个随机密钥，完成了域名的注册。name_new命令返回的两个字符串包括一串哈希值和一个随机密钥（本例中是a05555e0fc56c023），用于将域名公开发布。一旦注册的域名被记录到域名币区块链中，就可以通过name_firstupdate命令并提供随机密钥，将其转换为公开注册。

```
$ namecoind name_firstupdate d/mastering-bitcoin a05555e0fc56c023 "{"map":
{"www": {"ip":"1.2.3.4"}}}"
b7a2e59c0a26e5e2664948946ebeca1260985c2f616ba579e6bc7f35ec234b01
```

这个例子将域名www.mastering-bitcoin.bit映射到IP地址*1.2.3.4*。返回的哈希是一个交易ID，可用于跟踪注册过程。你可以使用name_list命令查看名下已注册的域名。

```
$ namecoind name_list

[
    {
        "name" : "d/mastering-bitcoin",
        "value" : "{map: {www: {ip:1.2.3.4}}}}",
        "address" : "NCccBXrRUahAGrisBA1BLPWQfSrups8Geh",
        "expires_in" : 35929
    }
]
```

每隔36000个区块（大约200到250天），域名币注册的名称就需要更新一次。name_update命令的运行不需要费用，也就是说，在域名币系统中域名的续期是免费的。也有第三方供应商通过web接口提供代理注册、自动续期、更新服务，收取一点费用。通过第三方供应商可以避免自己运行一个域名币客户端，但是也失去了域名币提供的去中心化域

名注册服务的独立控制权。

比特信

比特信（Bitmessage）是一个实现了安全消息服务的比特币替代链，本质上它其实就是一种无须中央服务器的加密电子邮件系统。用户可以通过比特信地址给其他用户发送消息。消息的操作方式与比特币交易类似，主要区别在于消息不是持久保存的，超过两天的消息将不再发送给接收方，也就是说，两天后消息就找不到了。发送方和接收方都是匿名的，除了比特信地址没有其他身份信息，但是收发双方都是经过严格验证的，不会出现“欺骗”消息。比特信是加密后发送给接收方的，因此可以避免被监听。监听者只有侵入接收方的设备，才有可能截获消息。

以太坊

以太坊（Ethereum）是一个基于区块链账本的、图灵完备的合约处理和运行平台。它不是比特币的克隆版，而是一个完全独立的设计和应用。以太坊自带的内置货币，叫作以太币（**ether**），运行合约时需要使用以太币。以太坊区块链记录的东西叫合约（**contracts**），这些合约以一种底层的、类似字节码的、图灵完备的编程语言来描述。本质上，一个合约就是一个在以太坊系统所有节点上运行的程序。以太坊合约可以存储数据，发送和接收以太币，存储以太币，执行无穷范围（因此是图灵完备的）的计算动作，充当去中心化自治软件系统的代理。

以太坊可以实现相当复杂的系统，甚至可以用于实现其他的替代链。举例说明，下面是使用以太坊实现的一个类似域名币的域名注册合约（或者更确切地说，是用可编译为以太坊代码的高级语言编写的）。

```
if !contract.storage[msg.data[0]]: # Is the key not yet taken?
    # Then take it!
    contract.storage[msg.data[0]] = msg.data[1]
    return(1)
else:

    return(0) // Otherwise do nothing
```

替代币的未来

总体来说，加密货币的未来比比特币更加光明。比特币引入的全新的去中心化组织和共识机制，已经成功地衍生出了几百项不可思议的创新。这些创新很有可能影响社会的不同领域，从分布式系统科学到金融、经济、货币、中央银行、企业治理等。很多原本要求中心机构或组织作为权威或信用点进行控制的人类行为，现在都可以实现去中心化了。区块链和共识系统的发明将大大降低大型系统的组织和协调成本，同时消除了权力集中、腐败和逃避监管的隐患。

第10章 比特币安全

保证比特币的安全是一项挑战，因为比特币不像银行账户的余额，不是价值的抽象体现。比特币与数字现金或黄金很相像。俗话说：“现实占有，十诉九胜。”在比特币的情形中，这个谚语可以改成“现实占有，十诉十胜。”拥有密钥解锁比特币与拥有现金或贵金属是等价的。你有可能把它们弄丢，放错地方，还有可能被盗，或者不小心给错了金额。不管发生哪种情况，资金都是没法追回来的。正如你把现金丢到了马路上，被他人捡走就很难找回了。

但是，比特币也有现金、黄金或银行账户不具备的功能。存放密钥的比特币钱包可以像任何普通文件一样进行备份；可以存储多个复制文件，甚至可以打印在纸张上进行备份。你肯定无法“备份”现金、黄金或者银行账户。比特币与普通货币的差异如此巨大，我们不得不用一种新的思路来考虑其安全性问题。

安全原则

比特币的核心原则是去中心化，这一点对其安全性非常重要。在诸如传统银行或支付网络等中心化的模型中，安全依赖于访问控制和审查，这可以把“坏蛋”挡在系统之外。与之相比，在一个类似比特币的去中心化系统中，维护系统安全的责任及控制权都交给了用户。因为比特币网络的安全是基于工作量证明，而不是访问控制，因此网络可以是开放的，数据流量本身也不需要进行加密。

在传统的支付网络——比如信用卡系统中，支付是开放式的，因为交易本身包含了用户的私人身份信息（信用卡账号）。第一次刷卡完成后，任何能访问卡信息的人都可以一次又一次地从账户中“取”资金，向用户收费。因此，支付网络必须对端到端的通信进行加密，确保监听者或中间环节无法在支付数据传输或存储的过程中截获交易信息。如果一个坏人获得了访问系统的权限，他就可以截获当前的交易信息和支付令牌，这些信息可以用于伪造交易。更糟糕的是，一旦客户信息泄露，客户的个人身份信息就会暴露在盗窃者的眼皮底下。客户这时必须立即采取措施，以防失窃账户被盗窃者用于欺诈行为。

比特币则完全不同。一个比特币交易仅仅授权向特定接收者发送特定价值的比特币，交易无法伪造也不能修改。这个过程不会泄露任何个人信息，比如当事人身份等，也无法用于授权其他支付交易。因此，比特币支付网络不需要进行加密，也不需防范窃听。实际上，你可以在诸如不安全的WiFi或者蓝牙等开放的公共网络上传播比特币交易信息，而比特币交易的安全性并不会因此而降低。

比特币的去中心化安全模型将大量权力交到了用户手中。伴随着权力而来的是维护密钥安全的责任。对大多数用户而言，做到这一点并不

容易，特别是在诸如联网的智能手机或笔记本电脑等通用用途的计算设备上。虽然比特币的去中心化模型防止了类似信用卡那样大量用户信息泄露的风险，但还是有很多比特币用户因无法保证其密钥安全，导致账户被黑客盗取。

开发安全的比特币系统

对比特币开发者来说，最重要的原则就是去中心化。开发者大都了解中心化的安全模型，也希望将这些模型应用到比特币应用程序中，但最终都是无功而返。

比特币的安全依赖于密钥的去中心化控制和矿工们独立的交易验证。如果想充分利用比特币的安全特性，就必须确保不会脱离比特币的安全模型。简单地说就是：不要让用户失去对密钥的控制权，不要让交易脱离区块链。

举例来说，很多早期的比特币交易系统将所有用户的资金集中在一个存储私钥的“热”钱包软件内，而钱包数据则存储在单一的服务器上。这样一种设计，剥夺了用户的控制权，并将代表控制权的密钥集中存储在一台服务器上。很多类似的系统都被黑客攻击过，给客户带来了灾难性的损失。

另外一个常见的错误行为是进行“离链”交易，试图利用这种方式降低交易费用或提高交易处理效率。“离链”系统会在其内部的中心化账本中记录交易信息，仅仅不定期地与比特币区块链进行同步。这种做法再次将去中心化的比特币安全机制，替换为专有的中心化方式。当交易离开区块链时，未做好安全防范措施的中心化账本可能会在不知不觉间就被篡改，并导致资金被转移、占用。

除非你打算投入大量财力、物力到运营安全上，比如进行多重的访

问控制、严格的事后审计（正如传统银行所做的那样），想将资金从比特币的去中心化安全场景中剥离出来，都必须慎之又慎。实际上，即使你有足够的资金和条件去创建一个强大的安全模型，这种设计也仅仅是复制了一个脆弱的、账户易受窃取的传统金融网络，深受贪污和挪用的困扰。要想充分利用比特币特有的去中心化安全模型，你必须抵御中心化架构带来的诱惑，虽然这种架构令人感到熟悉，但它终将摧毁比特币的安全性。

信任根

传统安全架构是基于一种被称作信任根（**root of trust**）的概念，它是用作系统或应用安全基础的可信核心。安全架构围绕着信任根开发，如同一系列同心圆，或者层层包裹的洋葱，围绕着中心逐层向外扩展。每一层都建立在更加可信的内层之上，方法包括访问控制、数字签名、加密，以及其他安全手段。软件系统愈加复杂，其隐含bug（漏洞）的可能性也愈高，安全威胁也随之增多。结果就是系统越复杂，安全越难确保。信任根的概念就是保证大多数信任被置于系统复杂度最低的部分，也是系统中安全风险最小的部分，更复杂的软件则在其上构建。这种安全架构可以在不同规模的系统中重复使用，首先在单一系统的硬件层建立一个信任根，然后不断向上扩展信任根，先是操作系统，然后是更高层的系统服务，最终形成了一个个围绕着信任根、信任递减的同心圆。

比特币的安全架构与之不同。在比特币中，共识系统创建了一个可信的并且完全去中心化的公共账本。一个得到正确验证的区块链使用创世区块作为信任根，创建了一条一直延伸到当前区块的信任链。比特币系统可以也必须使用区块链作为它们的信任根。当设计一个复杂的、需要在多个系统上提供服务的比特币应用时，你需要非常仔细地去审查其安全架构，以弄清信任放置在何处。最终，唯一需要明确信任的是一条

完整验证过的区块链。如果应用程序显性或隐性地依赖于除了区块链之外的任何东西，就需要引起关注，因为这可能就是系统最脆弱之处。一个评估应用安全架构的好方法是单独考量每个组件，设想该组件被完全攻破并被入侵者控制的场景。依次取出应用程序的每个组件，评估它被破坏时对整体安全造成的影响。如果你的应用程序在该组件被破坏时不再安全，那就说明你不该将信任放置在这个组件中。一个没有漏洞的比特币应用程序应该只受限于比特币的共识机制，也就是说，其信任根是基于比特币系统安全架构最坚固的部分。

大量被黑客攻击的比特币交易的例子说明了这一点的重要性，因为这些系统的安全架构和设计连最基本的检测都无法通过。这些中心化的实现方式将信任置于比特币区块链之外的组件上，比如热钱包、中心化的账本数据库、脆弱的加密密钥，以及类似的其他方案。

用户安全的最佳实践

人类已经使用了几千年的物理安全控制手段。相对地，我们在数字安全上的经验还不到50年。现代通用用途的操作系统并不十分安全，且不适合保存数字货币。我们的计算机因保持与互联网的连接状态而持续地暴露在外部的威胁当中。这些计算机上运行着不同的人所编写的成百上千的软件组件，这些组件通常都可以不受限制地访问用户的文件。在这些数量众多的软件当中，只要存在一个“流氓软件”，就能截获你的键盘输入，非法入侵文件，盗取钱包软件中保存的比特币。要达到免受病毒、木马的影响，所需的计算机维护水平已超过了大多数用户所具有的技能。

虽然信息安全的研究和发展已进行了几十年，但是数字资产仍然极易受到攻击。即使受到最高等级防护和限制的系统，比如金融服务企业、情报机构、防务承包商，也经常受到攻击。比特币创建了带有固有价值的数字资产，这些资产被窃取时可以立即并不可撤销地转移到新的账户，这对黑客来说是一个巨大的刺激。直到现在，黑客都是通过盗取身份信息或者账户令牌来获取资产，比如信用卡、银行账号等。尽管金融信息的掩饰和洗白越来越困难，但是盗窃行为也在不断升级。比特币的出现加剧了这个问题，因为它不需要掩饰和洗白，比特币本身就是拥有内在价值的数字资产。

幸运的是，比特币也创造了一种提高系统安全的激励机制。相对之前模糊和间接的计算机风险，比特币使这些风险更加清晰明显。在机器上保存比特币，使用户将其注意力集中到提高机器安全性的需求上来。结果就是比特币和其他数字货币，在越来越多的领域被接受并使用。我们看到，不管是黑客技术还是安全解决方案都得到了提升。简单地说，黑客现在有了一个非常诱人的目标，而用户也有了清晰的认识来保护自

己。

在过去的3年中，随着比特币不断被接纳，一个直接结果就是在信息安全领域取得了一系列巨大的创新，比如硬件加密、密钥存储和硬件钱包、多重签名技术、数字契约等。在接下来的内容中，我们将讨论几种实用的用户安全最佳实践。

物理比特币存储

相对信息安全，大多数用户对物理安全都感觉更加顺手，因此一种非常有效的保护比特币安全的方法就是将信息转化为物理形式。比特币密钥仅仅是一串长长的数字。这意味着它们可以以物理的形式进行存储，比如打印到纸上或者刻制到硬币上。保证密钥安全就转变成简单地物理保护打印的比特币密钥的安全。一系列打印在纸张上的密钥被称作“纸钱包”，很多免费的工具都可以用来打印纸钱包。我自己的做法是将绝大多数（99%以上）比特币保存在使用BIP0038进行加密的纸钱包上，这些纸钱包可以复制多份，锁进保险柜。将比特币保存在线下的方式叫作“冷存储（**cold storage**）”，是最高效的安全技术之一。冷存储系统是一种通过离线系统（永远不与互联网相连的系统）生成密钥，并将其离线存储在纸张、数字介质（比如U盘）的系统。

硬件钱包

从长远来看，比特币安全会越来越多地以硬件防篡改钱包的形式出现。不像智能电话或者桌面电脑，比特币的硬件钱包只有一个目的：安全地存储比特币。由于没有可被攻击的通用用途软件，仅提供有限接口，硬件钱包可以为非专业用户提供几乎万无一失的安全保护。我希望能看到硬件钱包成为比特币存储的主要形式。Trezor是这类硬件钱包的

一个例子（**http://www.bitcointrezor.com/**）。

平衡风险

虽然大多数用户都能正确认识比特币遭遇盗窃的风险，但是还有一个更大的风险——数据文件随时都可能丢失。如果文件中包含比特币，一旦丢失就会更加痛苦。在保护比特币钱包的努力中，用户必须非常小心，以免走过头导致比特币丢失。在2010年夏，一个著名的比特币认知和教育项目就丢失了将近7000比特币。他们在防范盗窃的过程中，实施了一系列复杂的加密备份，但是最终他们却把加密的密钥给弄丢了，备份变得一文不值，财富也白白丢失了。如果保护措施过头，就像把钱币埋在沙漠中，虽然财产安全了，但也有可能永远都无法再找回来了。

分散风险

你会将全部家当换成现金、放在钱包里随身携带吗？大多数人都会认为这是鲁莽的行为，但是比特币用户却经常将他们的全部比特币放在同一个钱包软件里。实际上，用户应该将比特币存放在多个不同的钱包中，从而达到分散风险的目的。明智的用户只会将小额比特币存放在在线或移动钱包中当作零钱使用，金额一般小于全部比特币金额的5%。剩下的部分则拆分后存储在不同的介质中，比如电脑钱包、离线钱包（冷存储）等。

多重签名和治理

当公司或个人存储大量比特币时，他们最好考虑使用多重签名比特

币地址。使用多重签名地址，用户支付时必须同时提供多个签名，这很好地保护了资金的安全。签名密钥必须存储在几个不同的地点，并由不同的人进行控制。比如，在公司环境中，密钥必须由多名公司高管独立生成并分别保管，确保没有一个人可以单独对资金安全造成威胁。多重签名在某些情况下，比如在一人拥有多个地址并且存储在不同地点时，也能提供冗余的功能。

生存性问题

一个重要的却经常被忽视的安全考虑是可用性，特别是在密钥拥有者失能或死亡的情况下。比特币用户被告知应该使用复杂的密码，并将其密钥存放在安全和隐蔽的地方，不跟任何人分享。不幸的是，当用户无法对比特币进行解锁时，他的家庭成员也无法恢复任何资金的使用权力。实际上，在大多数情况下，比特币用户的家庭可能根本就不知道这笔比特币资金的存在。

如果拥有很多比特币，你应该考虑与可信任的亲属或者律师共享解密资金的细节。也可以与专业的“数字资产执行者”的律师一起设置多重签名，做好遗产规划，从而制订更为复杂的生存性方案。

总结

比特币是一个全新的、史无前例的、复杂的技术。随着时间的推移，我们将开发出更好的安全工具，研究出更便于非专业人士使用的方法。现在，比特币用户可以使用这里讨论的一些技巧，享受安全无忧的比特币体验。

附录A 交易脚本语言操作符、常量、符号

表A.1列出了将数值压入栈的操作符。

表A.1　入栈操作

符号	值（十六进制）	描述
OP_0 或 OP_FALSE	0x00	将一个空数组压入栈中
1 –75	0x01 –0x4b	将接下来的 N 字节入栈，其中 N 大小为 1 到 75 字节
OP_PUSHDATA1	0x4c	下 1 个字节代表长度 N，将接下来的 N 个字节入栈
OP_PUSHDATA2	0x4d	下 2 个字节代表长度 N，将接下来的 N 个字节入栈
OP_PUSHDATA4	0x4e	下 4 个字节代表长度 N，将接下来的 N 个字节入栈
OP_1NEGATE	0x4f	将“－1”压入栈中
OP_RESERVED	0x50	终止——除非在未执行的 OP_IF 语句中出现，交易无效
OP_1 或 OP_TRUE	0x51	将“1”入栈
OP_2 或 OP_16	0x52 到 0x60	对于 OP_N，将“N”入栈，比如 OP_2 就将“2”入栈

表A.2列出了条件控制操作符。

表A.2　条件控制

符号	值（十六进制）	描述
OP_NOP	0x61	空操作
OP_VER	0x62	终止——除非在未执行的 OP_IF 语句中出现，交易无效
OP_IF	0x63	如果栈顶不为 0，执行语句
OP_NOTIF	0x64	如果栈顶为 0，执行语句
OP_VERIF	0x65	终止——交易无效
OP_VERNOTIF	0x66	终止——交易无效
OP_ELSE	0x67	只有前面的条件语句未被执行，才会执行
OP_ENDIF	0x68	OP_IF，OP_NOTIF，OP_ELSE 代码块结束标志
OP_VERIFY	0x69	检查栈顶，如果不为 TRUE 则终止或交易无效
OP_RETURN	0x6a	终止，交易无效

表A.3列出了控制栈的操作符。

表A.3　栈操作

符号	值（十六进制）	描述
OP_TOALTSTACK	0x6b	将主栈顶元素取出放入辅助栈
OP_FROMALTSTACK	0x6c	从辅助栈栈顶取出元素放入主栈
OP_2DROP	0x6d	取出栈顶 2 个元素
OP_2DUP	0x6e	复制栈顶 2 个元素
OP_3DUP	0x6f	复制栈顶 3 个元素
OP_2OVER	0x70	复制第 3 和第 4 个元素到栈顶
OP_2ROT	0x71	将第 5 和第 6 个元素移动到栈顶
OP_2SWAP	0x72	将栈顶的两对元素互换
OP_IFDUP	0x73	如果栈顶元素不为 0，则复制该元素
OP_DEPTH	0x74	计算栈中的元素数量，并将结果压入栈顶
OP_DROP	0x75	将栈顶元素出栈
OP_DUP	0x76	复制栈顶元素

（续表）

符号	值（十六进制）	描述
OP_NIP	0x77	将第 2 个元素出栈
OP_OVER	0x78	复制栈的第 2 个元素，并压入栈顶
OP_PICK	0x79	将栈顶元素 N 出栈，然后复制第 N 个元素到栈顶
OP_ROLL	0x7a	将栈顶元素 N 出栈，然后移动第 N 个元素到栈顶
OP_ROT	0x7b	将栈顶的 3 个元素翻转
OP_SWAP	0x7c	将栈顶的 3 个元素互换
OP_TUCK	0x7d	将栈顶元素复制并插入到第 1 和第 2 个元素之间

表A.4列出了字符串操作符。

表A.4　字符串拼接操作符

符号	值（十六进制）	描述
OP_CAT	0x7e	禁用（连接顶上2个元素）
OP_SUBSTR	0x7f	禁用（返回子字符串）
OP_LEFT	0x80	禁用（返回左子串）
OP_RIGHT	0x81	禁用（返回右子串）
OP_SIZE	0x82	计算栈顶字符串长度，并将结果压入栈顶

表A.5列出了二进制运算和逻辑运算操作符。

表A.5　二进制运算和条件

符号	值（十六进制）	描述
OP_INVERT	0x83	禁用（将栈顶元素按位取反）
OP_AND	0x84	禁用（顶上2个元素进行“布尔与”）
OP_OR	0x85	禁用（顶上2元素“布尔或”）
OP_XOR	0x86	禁用（顶上2元素“布尔异或”）
OP_EQUAL	0x87	如果2元素相等，压入TURE（1），否则压入FALSE（0）
OP_EQUALVERIFY	0x88	与OP_EQUAL等效，如果不为TRUE，则之后运行OP_VERIFY
OP_RESERVED1	0x89	终止——交易无效，除非在未执行的OP_IF子句中出现
OP_RESERVED2	0x8a	终止——交易无效，除非在未执行的OP_IF子句中出现

表A.6显示了算术运算符。

表A.6　算术运算符

符号	值（十六进制）	描述
OP_1ADD	0x8b	顶部元素加 1
OP_1SUB	0x8c	顶部元素减 1
OP_2MUL	0x8d	禁用（顶部元素乘 2）
OP_2DIV	0x8e	禁用（顶部元素除 2）
OP_NEGATE	0x8f	顶部元素符号翻转
OP_ABS	0x90	顶部元素符号变为正
OP_NOT	0x91	如果顶部元素为 0 或 1，则取反；否则返回 0
OP_0NOTEQUAL	0x92	如果顶部元素为 0 则返回 0，否则返回 1
OP_ADD	0x93	顶部两元素出栈并相加，结果重新入栈
OP_SUB	0x94	顶部两元素出栈，将第 2 个减第 1 个的差入栈
OP_MUL	0x95	禁用（顶部两元素相乘）
OP_DIV	0x96	禁用（第 2 个元素除以第 1 个元素的商）
OP_MOD	0x97	禁用（第 2 个元素除以第 1 个元素的余数）
OP_LSHIFT	0x98	禁用（第 2 个元素左移，左移位数为第 1 个元素的数值）
OP_RSHIFT	0x99	禁用（第 2 个元素右移，右移位数为第 1 个元素的数值）
OP_BOOLAND	0x9a	顶上 2 个元素“布尔与”
OP_BOOLOR	0x9b	顶上两元素“布尔或”
OP_NUMEQUAL	0x9c	如果顶上两元素相等，返回 TRUE
OP_NUMEQUALVERIFY	0x9d	与 NUMEQUAL 相同，如果不是 TRUE，进行 OP_VERIFY

（续表）

符号	值（十六进制）	描述
OP_NUMNO TEQUAL	0x9e	如果顶上两元素不等，返回 TRUE
OP _ LESST-HAN	0x9f	如果第 2 个元素比第 1 个小，返回 TRUE
OP_GREAT ERTHAN	0xa0	如果第 2 个元素比第 1 个大，返回 TRUE
OP_LESSTHA NOREQUAL	0xa1	如果第 2 个元素小于等于第 1 个元素，返回 TRUE
OP_GREATERT HANOREQUAL	0xa2	如果第 2 个元素大于等于第 1 个元素，返回 TRUE
OP_MIN	0xa3	返回顶上两个元素中较小的一个
OP_MAX	0xa4	返回顶上两个元素中较大的一个
OP_WITHIN	0xa5	如果第 3 个元素介于第 2 个和第 1 个之间，返回 TRUE

表A.7显示的是加密函数操作符。

表A.7　加密和哈希操作符

符号	值（十六进制）	描述
OP_RIPEMD160	0xa6	返回顶部元素的 RIPEMD160 哈希
OP_SHA1	0xa7	返回顶部元素的 SHA1 哈希
OP_SHA256	0xa8	返回顶部元素的 SHA256 哈希
OP_HASH160	0xa9	返回顶部元素的 RIPEMD160［SHA256（x）］哈希
OP_HASH256	0xaa	返回顶部元素的 SHA256［SHA256（x）］哈希
OP_CODESEPARATOR	0xab	标记已进行签名验证的数据的开始处
OP_CHECKSIG	0xac	将公钥和签名出栈，并验证交易的签名，如果匹配返回 TRUE
OP_CHECKSIGVERIFY	0xad	与 CHECKSIG 相同，如果结果不为 TRUE 则执行 OP_VERIFY

（续表）

符号	值（十六进制）	描述
OP_CHECKMULTISIG	0xae	对每对签名和公钥执行 CHECKSIG，所有的签名和公钥都必须能够匹配，因为如果存在 BUG，一个未使用的外部值会从堆栈中删除。采用 OP_NOP 作为替代方案
OP_CHECKMULTISIGVERIFY	0xaf	与 CHECKMULTISIG 相同，如果结果不为 TRUE，则执行 OP_VERIFY

表A.8显示空操作符。

表A.8　空操作符

符号	值（十六进制）	描述
OP_NOP1-OP_NOP10	0xb0 –0xb9	不做任何操作，忽略

表A.9显示保留操作符，仅供内部脚本解析使用。

表A.9　保留操作符，内部解析使用

符号	值（十六进制）	描述
OP_SMALLDATA	0xf9	表示小数据域
OP_SMALLINTEGER	0xfa	表示小整数数据域
OP_PUBKEYS	0xfb	表示多个公钥域
OP_PUBKEYHASH	0xfd	表示公钥哈希域
OP_PUBKEY	0xfe	表示公钥域
OP_INVALIDOPCODE	0xff	表示任何尚未指定的操作符

附录B 比特币改进提案

比特币改进提案（Bitcoin improvement proposals，简称BIP）是一种向比特币社区提供信息，描述比特币及其处理过程，以及外部环境新特性的设计文档。

基于BIP0001《**BIP**的目的和指南》，BIP可以分为3种类型。

标准**BIP**（**Standard BIP**）

描述影响大多数或全部比特币实现的变化，比如网络协议的改变，区块或交易验证规则的改变，或者任何影响比特币应用互操作性的变化或补充。

说明性**BIP**（**Informational BIP**）

说明性BIP不提出新的特性，仅描述比特币的设计问题，向比特币社区提供通用指南或信息。说明性BIP不需要取得比特币社区的同意或推荐，所以用户或开发者可以忽略说明性BIP，也可以接受其中的建议。

过程**BIP**（**Process BIP**）

描述比特币的处理过程，或者提议对过程进行改进或添加新的事件。过程BIP与标准BIP类似，但适用领域不包括比特币协议本身。它们可能提出一种实现方案，但不会针对比特币代码库；过程BIP一般都要求社区达成共识；不像说明性BIP，过程BIP是强制性的，用户不能选择忽略。过程BIP的示例包括决策过程的规程、指南及变化，比特币开发中工具或环境的变化等。任何元BIP（meta-BIP）也被视为过程BIP。

比特币改进提案记录于GitHub版本库中（**http://github.com/bitcoin/bips**）。表B.1显示了截至2014年秋天的所有BIP的快照。要了解已有BIP及其内容的最新消息，可以访问官方文档库。

表B.1　BIP快照

BIP序号	链接	标题	作者	类型	状态
1	*https://github.com/bitcoin/bips/blob/master/bip-0001.mediawiki*	BIP目标与指南	Amir Taaki	标准	活跃
10	*https://github.com/bitcoin/bips/blob/master/bip-0010.mediawiki*	多重签名交易分布	Alan Reiner	说明性	草案
11	*https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki*	M-of-N标准交易	Gavin Andresen	标准	采纳
12	*https://github.com/bitcoin/bips/blob/master/bip-0012.mediawiki*	OP_EVAL	Gavin Andresen	标准	撤销
13	*https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki*	支付到脚本哈希地址格式	Gavin Andresen	标准	终稿
14	*https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki*	协议版本与用户代理	Amir Taaki，Patrick Strateman	标准	采纳
15	*https://github.com/bitcoin/bips/blob/master/bip-0015.mediawiki*	别名	Amir Taaki	标准	撤销
16	*https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki*	支付到脚本哈希	Gavin Andresen	标准	采纳
17	*https://github.com/bitcoin/bips/blob/master/bip-0017.mediawiki*	OP_CHECKHASHVERIFY（CHV）	Luke Dashjr	撤销	草案
18	*https://github.com/bitcoin/bips/blob/master/bip-0018.mediawiki*	哈希脚本校验	Luke Dashjr	标准	草案
19	*https://github.com/bitcoin/bips/blob/master/bip-0019.mediawiki*	M-of-N标准交易（Low SigOp）	Luke Dashjr	标准	草案
20	*https://github.com/bitcoin/bips/blob/master/bip-0020.mediawiki*	URI方案	Luke Dashjr	标准	被替换
21	*https://github.com/bitcoin/bips/blob/master/bip-0021.mediawiki*	URI方案	Nils Schneider，Matt Corallo	标准	采纳
22	*https://github.com/bitcoin/bips/blob/master/bip-0022.mediawiki*	getblocktemplate—基础	Luke Dashjr	标准	采纳

（续表）

BIP 序号	链接	标题	作者	类型	状态
23	*https://github.com/bitcoin/bips/blob/master/bip-0023.mediawiki*	getblocktemplate—矿池挖矿	Luke Dashjr	标准	采纳
30	*https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki*	重复交易	Pieter Wuille	标准	采纳
31	*https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki*	应答消息	Mike Hearn	标准	采纳
32	*https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki*	层次化确定性钱包	Pieter Wuille	说明性	采纳
33	*https://github.com/bitcoin/bips/blob/master/bip-0033.mediawiki*	分层节点	Amir Taaki	标准	草案
34	*https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki*	区块版本 2，币基中的高度	Gavin Andresen	标准	采纳
35	*https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki*	内存池消息	Jeff Garzik	标准	采纳
36	*https://github.com/bitcoin/bips/blob/master/bip-0036.mediawiki*	用户服务	Stefan Thomas	标准	草案
37	*https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki*	布隆过滤器	Mike Hearn and Matt Corallo	标准	采纳
38	*https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki*	密码保护的私钥	Mike Caldwell	标准	草案
39	*https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki*	用于创建确定性密钥的助记码	Slush	标准	草案
40	—	Stratum 连线协议	Slush	标准	BIP 编号已分配

（续表）

BIP 序号	链接	标题	作者	类型	状态
41	—	Stratum 挖矿协议	Slush	标准	BIP 编号已分配
42	*https://github.com/bitcoin/bips/blob/master/bip-0042.mediawiki*	比特币的有限货币供应	Pieter Wuille	标准	草案
43	*https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki*	确定性钱包的目标域	Slush	标准	草案
44	*https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki*	确定性钱包的多账户层次	Slush	标准	草案
50	*https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki*	2013 年 3 月链分叉事后研究	Gavin Andresen	说明性	草案
60	*https://github.com/bitcoin/bips/blob/master/bip-0060.mediawiki*	固定长度“版本”消息（交易中继字段）	Amir Taaki	标准	草案
61	*https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki*	“拒绝”P2P 消息	Gavin Andresen	标准	草案
62	*https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki*	处理延展性	Pieter Wuille	标准	草案
63	—	隐形地址	Peter Todd	标准	BIP 编号已分配
64	*https://github.com/bitcoin/bips/blob/master/bip-0064.mediawiki*	获取 utxos 消息	Mike Hearn	标准	草案

（续表）

BIP 序号	链接	标题	作者	类型	状态
70	*https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki*	支付协议	Gavin Andresen	标准	草案
71	*https://github.com/bitcoin/bips/blob/master/bip-0071.mediawiki*	支付协议 MIME 类型	Gavin Andresen	标准	草案
72	*https://github.com/bitcoin/bips/blob/master/bip-0072.mediawiki*	支付协议的 URIs	Gavin Andresen	标准	草案
73	*https://github.com/bitcoin/bips/blob/master/bip-0073.mediawiki*	在支付请求 URIs 中使用“Accept”报文头	Stephen Pair	标准	草案

附录C pycoin、ku和tx

pycoin库（http：//github.com/richardkiss/pycoin）最初是由理查德·吉斯（Richard Kiss）编写并维护的，是一个基于Python的库，支持比特币密钥和交易的操作。由于对脚本语言的支持效果很好，它甚至可以正确处理非标准交易。

pycoin库支持Python2（2.7.x）和Python3（3.3以上版本），并随库附带了一些好用的命令行工具，如ku和tx。

实用密钥工具（**Key Utility，KU**）

命令行工具ku（key utility）是一把操作密钥的“瑞士军刀”。它支持BIP32密钥，WIF和地址（比特币地址或替代币地址）。以下是一些范例。

使用GPG的默认熵源及系统随机源**/dev/random**创建一个BIP32密钥。

```
$ ku create

input           : create
network         : Bitcoin
wallet key      : xprv9s21ZrQH143K3LU5ctPZTBnb9kTjA5Su9DcWHvXJemiJB-
sY7VqXUG7hipgdWaU
                   m2nhnzdvxJf5KJo9vjP2nABX65c5sFsWsV8oXcbpehtJi
public version  : xpub661MyMwAqRbcFpYYiuvZpKjKhnJDZYAkW-
```

```
SY76JvvD7FH4fsG3Nqiov2CfxzxY8
                        DGcpfT56AMFeo8M8KPkFMfLUtvwjwb6WPv8rY65L2q8Hz
tree depth       : 0
fingerprint      : 9d9c6092
parent f'print   : 00000000
child index      : 0
chain code       :
80574fb260edaa4905bc86c9a47d30c697c50047ed466c0d4a5167f6821e8f3c
private key      : yes
secret exponent :
112471538590155650688604752840386134637231974546906847202389294096567806844862
 hex             :
f8a8a28b28a916e1043cc0aca52033a18a13cab1638d544006469bc171fddfbe
wif              : L5Z54xi6qJusQT42JHA44mfPVZGjyb4XBRWfxAzUWwRiGx1kV4sP
 uncompressed    : 5KhoEavGNNH4GHKoy2Ptu4KfdNp4r56L5B5un8FP6RZnbsz5Nmb
public pair x    :
76460638240546478364843397478278468101877117767873462127021560368290114016034
public pair y    :
59807879657469774102040120298272207730921291736633247737077406753676825777701
 x as hex        :
a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
 y as hex        :
843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
y parity         : odd
key pair as sec :
03a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
 uncompressed    :
04a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322

843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
hash160          : 9d9c609247174ae323acfc96c852753fe3c8819d
 uncompressed    : 8870d869800c9b91ce1eb460f4c60540f87c15d7
Bitcoin address : 1FNNRQ5fSv1wBi5gyfVBs2rkNheMGt86sp
 uncompressed    : 1DSS5isnH4FsVaLVjeVXewVSpfqktdiQAM
```

使用口令创建BIP32密钥如下。

本例中的口令太简单了，很容易被猜到。

```
$ ku P:foo

input            : P:foo
network          : Bitcoin
wallet key       :
xprv9s21ZrQH143K31AgNK5pyVvW23gHnkBq2wh5aEk6g1s496M8ZMjxncCKZKgb5j
                    ZoY5eSJMJ2Vbyvi2hbmQnCuHBujZ2WXGTux1X2k9Krdtq
public version   : xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWo-
QayvtS
                    VYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy
tree depth       : 0
fingerprint      : 5d353a2e
parent f'print   : 00000000
child index      : 0
chain code       :
5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc
private key      : yes
secret exponent :
65825730547097305716057160437970790220123864299761908948746835886007793998275
 hex              :
91880b0e3017ba586b735fe7d04f1790f3c46b818a2151fb2def5f14dd2fd9c3
wif               : L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
 uncompressed     : 5JvNzA5vXDoKYJdw8SwwLHxUxaWvn9mDea6k1vRPCX7KLUVWa7W
public pair x     :
81821982719381104061777349269130419024493616650993589394553404347774393168191
public pair y     :
58994218069605424278320703250689780154785099509277691723126325051200459038290
 x as hex         :
b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
 y as hex         :
826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
y parity          : even
key pair as sec :
02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
 uncompressed     :
04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f

826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
hash160           : 5d353a2ecdb262477172852d57a3f11de0c19286
 uncompressed     : e5bd3a7e6cb62b4c820e51200fb1c148d79e67da
Bitcoin address : 19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
 uncompressed     : 1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT
```

以JSON格式获取消息如下。

```
$ ku P:foo -P -j
{
   "y_parity": "even",
   "public_pair_y_hex":
"826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52",
   "private_key": "no",
   "parent_fingerprint": "00000000",
   "tree_depth": "0",
   "network": "Bitcoin",
   "btc_address_uncompressed": "1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT",
   "key_pair_as_sec_uncompressed":
"04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f826d8b4d3010a
ea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52",
   "public_pair_x_hex":
"b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f",
   "wallet_key": "xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWo
QayvtSVYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy",
   "chain_code":
"5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc",
   "child_index": "0",
   "hash160_uncompressed": "e5bd3a7e6cb62b4c820e51200fb1c148d79e67da",
   "btc_address": "19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii",
   "fingerprint": "5d353a2e",
   "hash160": "5d353a2ecdb262477172852d57a3f11de0c19286",
   "input": "P:foo",
   "public_pair_x":
"81821982719381104061777349269130419024493616650993589394553404347774393168191",
   "public_pair_y":
"58994218069605424278320703250689780154785099509277691723126325051200459038290",
   "key_pair_as_sec":
"02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f"
}
```

BIP32公钥如下。

```
$ ku -w -P P:foo
xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtS-
VYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy
```

创建子密钥如下。

```
$ ku -w -s3/2 P:foo
xprv9wTErTSkjVyJa1v4cUTFMFkWMe5eu8ErbQcs9xajn-
sUzCBT7ykHAwdrxvG3g3f6BFk7ms5hHBvmbdutNmyg6iogWKxx6mefEw4M8EroLgKj
```

强化子密钥如下。

```
$ ku -w -s3/2H P:foo
xprv9wTErTSu5AWGkDeUPmqBcbZWX1xq85ZNX9iQRQW9DXwygFp7iRG-
Jo79dsVctcsCHsnZ3XU3DhsuaGZbDh8iDkBN45k67UKsJUXM1JfRCdn1
```

WIF如下。

```
$ ku -W P:foo
L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
```

地址如下。

```
$ ku -a P:foo
19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
```

批量创建子密钥如下。

```
$ ku P:foo -s 0/0-5 -w
xprv9xWkBDfyBXmZjBG9EiXBpy67KK72fphUp9utJokEBFtjsjiuKUUDF5V3TU8U8cDzytqYn-
Sekc8bYuJS8G3bhXxKWB89Ggn2dzLcoJsuEdRK
xprv9xWkBDfyBXmZnzKf3bAGifK593gT7WJZPnYAmvc77gUQVej5QHckc5Adtwxa28ACmA-
Ni9XhCrRvtFqQcUxt8rUgFz3souMiDdWxJDZnQxzx
xprv9xWkBDfyBXmZqdXA8y4SWqfBdy71gSW9sjx9JpCiJEiBwSMQyRxan6srXUPBtj3PTxQFkZJAi-
woUpmvtrxKZu4zfsnr3pqyy2vthpkwuoVq
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuq-
xYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p
xprv9xWkBDfyBXmZv2q3N66hhZ8DAcEnQDnXML1J62krJAcf7Xb1HJwuW2VMJQrCo-
fY2jtFXdiEY8UsRNJfqK6DAdyZXoMvtaLHyWQx3FS4A9zw
xprv9xWkBDfyBXmZw4jEYXU-
HYc9fT25k9irP87n2RqfJ5bqbjKdT84Mm7Wtc2xmzFuKg7iYf7XFHKkSsaYKWKJbR54bnyAD9GzjUY-
bAYTtN4ruo
```

创建相应的地址如下。

```
$ ku P:foo -s 0/0-5 -a
1MrjE78H1R1rqdFrmkjdHnPUdLCJALbv3x
1AnYyVEcuqeoVzH96zj1eYKwoWfwte2pxu
1GXr1kZfxE1FcK6ZRD5sqqqs5YfvuzA1Lb
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
1Cz2rTLjRM6pMnxPNrRKp9ZSvRtj5dDUML
1WstdwPnU6HEUPme1DQayN9nm6j7nDVEM
```

创建相应的WIF如下。

```
$ ku P:foo -s 0/0-5 -W
L5a4iE5k9gcJKGqX3FWmxzBYQc29PvZ6pgBaePLVqT5YByEnBomx
Kyjgne6GZwPGB6G6kJEhoPbmyjMP7D5d3zRbHVjwcq4iQXD9QqKQ
L4B3ygQxK6zH2NQGxLDee2H9v4Lvwg14cLJW7QwWPzCtKHdWMaQz
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF
L2oD6vA4TUyqPF8QG4vhUFSgwCyuuvFZ3v8SKHYFDwkbM765Nrfd
KzChTbc3kZFxUSJ3Kt54cxsogeFAD9CCM4zGB22si8nfKcThQn8C
```

通过选择BIP32字符串（与子密钥0/3对应），检查其是否起作用。

```
$ ku -W xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuq-
xYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF
$ ku -a xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuq-
xYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
```

嗯，看起来很熟悉。

从秘密指数创建如下。

```
$ ku 1

input            : 1
network          : Bitcoin
secret exponent : 1
 hex             : 1
wif              : KwDiBf89QgGbjEhKnhXJuH7LrciVrZi3qYjgd9M7rFU73sVHnoWn
 uncompressed    : 5HpHagT65TZzG1PH3CSu63k8DbpvD8s5ip4nEB3kEsreAnchuDf
public pair x    :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y    :
32670510020758816978083085130507043184471273380659243275938904335757337482424
 x as hex        :
79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 y as hex        :
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity         : even
key pair as sec :
0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 uncompressed    :
0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798

483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160          : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed    : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
 uncompressed    : 1EHNa6Q4Jz2uvNExL497mE43ikXhwF6kZm
```

莱特币版本如下。

```
$ ku -nL 1

input             : 1
network           : Litecoin
secret exponent   : 1
 hex              : 1
wif               : T33ydQRKp4FCW5LCLLUB7deioUMoveiwekdwUwyfRDeGZm76aUjV
 uncompressed     : 6u823ozcyt2rjPH8Z2ErsSXJB5PPQwK7VVTwwN4mxLBFrao69XQ
public pair x     :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y     :
32670510020758816978083085130507043184471273380659243275938904335757337482424
 x as hex         :
79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 y as hex         :
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity          : even
key pair as sec   :
0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 uncompressed     :
0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798

483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160           : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed     : 91b24bf9f5288532960ac687abb035127b1d28a5
Litecoin address  : LVuDpNCSSj6pQ7t9Pv6d6sUkLKoqDEVUnJ
 uncompressed     : LYWKqJhtPeGyBAw7WC8R3F7ovxtzAiubdM
```

狗币WIF如下。

```
$ ku -nD -W 1
QNcdLVw8fHkixm6NNyN6nVwxKek4u7qrioRbQmjxac5TVoTtZuot
```

从公钥（来自Testnet）创建如下。

```
$ ku -nT
55066263022277343669578718895168534326250603453777594175500187360389116729240,ev
en

input                    :
55066263022277343669578718895168534326250603453777594175500187360
                           389116729240,even
network                  : Bitcoin testnet
public pair x            :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y            :
32670510020758816978083085130507043184471273380659243275938904335757337482424
 x as hex                :
79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 y as hex                :
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity                 : even
key pair as sec          :
0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 uncompressed            :
0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798

483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160                  : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed            : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin testnet address : mrCDrCybB6J1vRfbwM5hemdJz73FwDBC8r
 uncompressed            : mtoKs9V381UAhUia3d7Vb9GNak8Qvmcsme
```

从hash 160创建如下。

```
$ ku 751e76e8199196d454941c45d1b3a323f1433bd6

input             : 751e76e8199196d454941c45d1b3a323f1433bd6
network           : Bitcoin
hash160           : 751e76e8199196d454941c45d1b3a323f1433bd6
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
```

作为狗币地址如下。

```
$ ku -nD 751e76e8199196d454941c45d1b3a323f1433bd6

input             : 751e76e8199196d454941c45d1b3a323f1433bd6
network           : Dogecoin
hash160           : 751e76e8199196d454941c45d1b3a323f1433bd6
Dogecoin address : DFpN6QqFfUm3gKNaxN6tNcab1FArL9cZLE
```

实用交易工具（Transaction Utility，TX）

命令行工具tx可以以人类可读的形式显示交易信息，从pycoin的交易缓存或web服务（blockchain.info, blockr.io, biteasy.com当前都能支持）获取原始交易，合并交易，添加或删除输入或输出，对交易进行签名。

下面是一些例子。

查看著名的“比萨”交易[PIZZA]。

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: consider setting environment variable PYCOIN_CACHE_DIR=~/.pycoin_cache
to cache transactions fetched via web services
warning: no service providers found for get_tx; consider setting environment
variable PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLOR-
ER
usage: tx [-h] [-t TRANSACTION_VERSION] [-l LOCK_TIME] [-n NETWORK] [-a]
          [-i address] [-f path-to-private-keys] [-g GPG_ARGUMENT]
          [--remove-tx-in tx_in_index_to_delete]
          [--remove-tx-out tx_out_index_to_delete] [-F transaction-fee] [-u]
          [-b BITCOIND_URL] [-o path-to-output-file]
          argument [argument ...]
tx: error: can't find Tx with id
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
```

哎呀！我们还没设置好web服务，我们先来做这个。

```
$ PYCOIN_CACHE_DIR=~/.pycoin_cache
$ PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
$ export PYCOIN_CACHE_DIR PYCOIN_SERVICE_PROVIDERS
```

这个设置不是自动完成的，所以命令行工具不会将你感兴趣的交易的私人信息泄露给第三方网站。如果你不在乎隐私，你可以将配置写到**.profile**文件中。

我们再试一遍。

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
Version:  1  tx hash
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a  159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
  0:                          (unknown) from
1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total output 10000000.00000 mBTC
including unspents in hex dump since transaction not fully signed
010000000141045e0ab2b0b82cde-
faf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a493046022100a7f26eda8749
31999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e9199238eb73f07c8f2
09504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff010010a5d4e80000001976a9147e
c1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

** can't validate transaction as source transactions missing
```

最后一行之所以会出现，是因为要验证一个交易的签名，从技术上需要其源交易。所以我们加上-a参数以使用交易源信息。

```
$ tx -a 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: transaction fees recommendations casually calculated and estimates may
be incorrect
warning: transaction fee lower than (casually calculated) expected value of 0.1
mBTC, transaction might not propogate
Version:  1  tx hash
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a  159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
```

```
  0: 17WFx2GQZUmh6Up2NDNCEDk3deYomdNCfk from
1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0
10000000.00000 mBTC  sig ok
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total input  10000000.00000 mBTC
Total output 10000000.00000 mBTC
Total fees        0.00000 mBTC

010000000141045e0ab2b0b82cde-
faf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a493046022100a7f26eda8749
31999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e9199238eb73f07c8f2
09504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff010010a5d4e8000000001976a9147e
c1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

all incoming transaction values validated
```

现在，我们来看一个特定地址的未花费输出（UTXO）。在区块1中，我们看到一个发送到地址12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX的铸币交易。使用fetch_unspent查找这个地址中的所有比特币。

```
$ fetch_unspent 12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX
a3a6f902a51a2cbebede144e48a88c05e608c2cce28024041a5b9874013a1e2a/
0/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/333000
cea36d008badf5c7866894b191d3239de9582d89b6b452b596f1f1b76347f8cb/
31/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/10000
065ef6b1463f552f675622a5d1fd2c08d6324b4402049f68e767a719e2049e8d/
86/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/10000
a66dddd42f9f2491d3c336ce5527d45cc5c2163aa-
ed3158f81dc054447f447a2/0/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/
10000
ffd901679de65d4398de90cefe68d2c3ef073c41f7e8dbec2fb5cd75fe71dfe7/0/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/100
d658ab87cc053b8dbcfd4aa2717fd23cc3edfe90ec75351fadd6a0f7993b461d/
5/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/911
36ebe0ca3237002acb12e1474a3859bde0ac84b419ec4ae373e63363ebef731c/
1/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/100000
fd87f9adebb17f4ebb1673da76ff48ad29e64b7afa02fda0f2c14e43d220fe24/0/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/1
dfdf0b375a987f17056e5e919ee6eadd87dad36c09c4016d4a03cea15e5c05e3/1/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/1337
cb2679bfd0a557b2dc0d8a6116822f3fcbe281ca3f3e18d3855aa7ea378fa373/0/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/1337
d6be34ccf6edddc3cf69842dce99fe503bf632ba2c2adb0f95c63f6706ae0c52/1/76a914119b098
e2e980a229e139a9ed01a469e518e6f2688ac/2000000

0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098/0/410496b538e85
3519c726a2c91e61ec11600ae1390813a627c66fb8be7947be63c52da7589379515d4e0a604f8141
781e62294721166bf621e73a82cbf2342c858eeac/5000000000
```

附录D　sx工具集可用命令

sx命令包括：

已过时的

Electrum风格的确定性密钥和地址。

genaddr 从钱包种子或主公钥中确定性地生成一个比特币地址。

genpriv 从种子中确定性地生成一个私钥。

genpub 从钱包种子或主公钥中确定性地生成一个公钥。

mpk 从确定性钱包种子中解开一个主公钥。

newseed 生成一个确定性钱包种子。

实验性的

应用

wallet 实验性命令行钱包。

离线区块链

区块头

showblkhead 显示区块头详细信息。

离线密钥和地址

基本

addr 查看一个公钥或私钥的比特币地址。

embed-addr 生成一个嵌入区块链的比特币地址。

get-pubkey 如果地址存在，取其公钥。

newkey 创建一个新私钥。

pubkey 查看私钥的公共部分。

validaddr 验证一个地址。

头脑存储

brainwallet 从任意口令生成一个256位的比特币私钥。

mnemonic 从128位electrum或bip32种子中生成12个助记单词。

HD/BIP32

hd-priv 从另一个HD私钥生成一个新的HD（层次化确定性）私钥。

hd-pub 从另一个HD私钥或公钥生成一个HD公钥。

hd-seed 生成一个新的随机HD密钥。

hd-to-address 将HD公钥或私钥转换为一个比特币地址。

hd-to-wif 将HD私钥转换为WIF格式私钥。

多重签名地址

scripthash 从原始脚本的十六进制数据创建BIP16脚本哈希地址。

隐形

stealth-addr 从给定输入查看一个隐形地址。

stealth-initiate 初始化一个新的隐形支付。

stealth-newkey 生成新的隐形密钥和地址。

stealth-show-addr 显示一个隐形地址的详细信息。

stealth-uncover 披露一个隐形地址。

stealth-uncover-secret 披露一个隐形秘密。

离线交易

脚本处理

mktx 创建一个未签名交易。

rawscript 从一个脚本创建原始十六进制表示。

set-input 设置交易输入。

showscript 显示一个原始脚本的详情。

showtx 显示交易详情。

sign-input 对一个交易输入进行签名。

unwrap 从十六进制字符串中验证校验码，并恢复版本字节和原始数据。

validsig 验证交易输入的签名。

wrap 添加版本字节和校验码到十六进制字符串中。

在线（比特币**P2P**）

区块链更新

sendtx-node 将交易发送到一个节点上。

sendtx-p2p 将交易发送到比特币网络上。

在线（**BLOCKCHAIN.INFO**）

区块链查询（BLOCKCHAIN.INFO）

bci-fetch-last-height 使用*blockchain.info*获取最小区块高度。

bci-history　　从*blockchain.info*获取输出点、价值，以及其支出的列表。

区块链更新

sendtx-bci 发送交易到*blockchain.info/pushtx*.

在线（**BLOCKEXPLORER.COM**）

区块链查询（*blockexplorer.com*）

blke-fetch-transaction 从*blockexplorer.com*获取一个交易。

在线（**OBELISK**）

区块链查询

balance 以聪为单位显示一个地址的余额。

fetch-block-header 获取元素区块头。

fetch-last-height 获取最新区块高度。

fetch-stealth 使用网络连接向obelisk负载均衡器后台发送请求，以获取隐藏信息。

fetch-transaction 使用网络连接向obelisk负载均衡器后台发送请求，以获取原始交易信息。

fetch-transaction-index 获取交易所在区块的高度及交易位置索引号。

get-utxo 从给定的地址集中获取足够的未花费输出，以支付一定金额的比特币。

history 获取特定地址的输出点、价值及其支出的列表，使用grep过滤出未花费的输出，其结果可以被mktx命令调用。

validtx 验证一个交易。

区块链更新

sendtx-obelisk 发送交易到obelisk服务器。

区块链观察

monitor 监控一个地址。

watchtx 通过在网络中对一个特定哈希的搜索来观察交易。

OBELISK管理

initchain 初始化一个区块链。

实用工具

椭圆曲线计算

ec-add-modp 计算整数+整数的结果。

ec-multiply 整数与点的乘积。

ec-tweak-add 计算（点+整数×生成点）。

格式（BASE 58）

base58-decode 将base58转换为十六进制。

base58-encode 将十六进制转换为base58。

格式（BASE58CHECK）

base58check-decode 将base58check转换为十六进制。

base58check-encode 将十六进制转换为base58check。

decode-addr 将地址从base58check转换为内部RIPEMD表示。

encode-addr 将地址从内部RIPEMD表示转换为base58格式。

格式（WIF）

secret-to-wif 将秘密指数值转换为WIF格式。

wif-to-secret 将WIF格式转换为秘密指数值。

哈希

ripemd-hash 从标准输入数据通过RIPEMD转换为哈希。

sha256 采用SHA256算法对数据做哈希计算。

杂项

qrcode 离线生成比特币的二维码。

“聪”的换算

btc 将“聪”换算为比特币。

satoshi 将比特币换算成“聪”。

通过*sx help COMMAND*可以查看单个命令的详细说明。

接下来，我们将通过一些例子来尝试对密钥和地址进行操作。采用newkey命令，利用操作系统的随机数生成器生成一个新的私钥。我们将标准输出保存到文件**private_key**：

```
$ sx newkey > private_key
$ cat private_key
5Jgx3UAaXw8AcCQCi1j7uaTaqpz2fqNR9K3r4apxdYn6rTzR1PL
```

现在，使用pubkey命令从刚才生成的私钥生成公钥。将私钥文件**private_key**通过重定向命令输出到标准输入，将标准输出重定向到一个新文件**public_key**：

```
$ sx pubkey < private_key > public_key
$ cat public_key
02fca46a6006a62dfdd2dbb2149359d0d97a04f430f12a7626dd409256c12be500
```

我们可以使用addr命令将公钥转换为地址。将公钥文件通过重定向输出到标准输入：

```
$ sx addr < public_key
17re1S4Q8ZHyCP8Kw7xQad1Lr6XUzWUnkG
```

生成的密钥被称为type-0非确定性密钥，意思是每个密钥均从随机数生成器生成。sx工具集也支持type-2确定性密钥，“主”密钥生成后，可以由它派生出一个子密钥链或子密钥树。

首先，我们创建一个“种子”，它将成为派生密钥链的基础，与Electrum钱包或其他类似的钱包兼容。我们使用newseed命令来生成种子：

```
$ sx newseed > seed
$ cat seed
eb68ee9f3df6bd4441a9feadec179ff1
```

种子的值也可以导出成容易阅读的助记码，这种形式比十六进制字符串更加容易保存和输入，使用的命令为mnemonic：

```
$ sx mnemonic < seed > words
$ cat words
adore repeat vision worst especially veil inch woman cast recall dwell appreci-
ate
```

助记词可以用于重新生成种子，使用的命令仍然是mnemonic：

```
$ sx mnemonic < words
eb68ee9f3df6bd4441a9feadec179ff1
```

利用这个种子，我们可以生成一系列的私钥和公钥，也可称之为密钥链。我们使用genpriv命令从种子来生成一系列的私钥，addr命令则用于生成对应的公钥：

```
$ sx genpriv 0 < seed
5JzY2cPZGViPGgXZ4Syb9Y4eUGjJpVt6sR8noxrpEcqgyj7LK7i
$ sx genpriv 0 < seed | sx addr
1esVQV2vR9JZPhFeRaeWkAhzmWq7Fi7t7

$ sx genpriv 1 < seed
5JdtL7ckAn3iFBFyVG1Bs3A5TqziFTaB9f8NeyNo8crnE2Sw5Mz
$ sx genpriv 1 < seed | sx addr
1G1oTeXitk76c2fvQWny4pryTdH1RTqSPW
```

基于确定性密钥，我们可以生成或重新生成上千个密钥，所有密钥均从确定性密钥链的单一种子衍生而来。这种技术已被多种钱包应用使用，用于生成可以利用几个简单的助记词进行备份和恢复的密钥。这要比每当一个新密钥生成时就要备份钱包中所有随机生成的密钥简单得多。

关于作者

安德烈亚斯·安东诺普洛斯（Andreas M. Antonopoulos）是一位著名的技术专家、连续创业者，现在他已经是比特币社区最著名、最受尊敬的人物之一。作为一位成功的公共演说家、教师和作家，安德烈亚斯擅长将复杂的问题变得简单以便于理解。作为一个顾问，他擅长帮助初创企业识别、评估、管理安全和业务风险。

安德烈亚斯的成长与互联网密不可分。青少年时期，他便在希腊的家中创办了他的首家公司，一个提供BBS服务的ISP。他毕业于世界排名前十的英国伦敦大学学院（UCL），获得了计算机科学、数据通信和分布式系统的学位。移居美国后，他与合伙人一起创办并管理了一家相当成功的技术研究公司。在此期间，他为数十位财富500强公司的高管提供网络、安全、数据中心和云计算方面的建议。在安全、云计算、数据中心等方面，安德烈亚斯撰写了200多篇文章，并在全世界范围内出版发行。同时，他还拥有两项关于网络和安全方面的专利。

从1990年起，安德烈亚斯开始在私人、专业和学术等各种场合讲授IT课题。从面向5个高管的小会议室到面对几千人的大会场，各种场合的演讲，磨炼了他的演讲技巧。他做过400多场的演讲，是公认的世界级的魅力演讲专家和教师。在2014年，他被聘为尼科西亚大学的教师，而尼科西亚是世界上第一个授予数字货币硕士学位的大学。任职期间，他协助开发了这门课程，还参与讲授介绍数字货币的网络公开课（MOOC）。

作为一个比特币企业家，安德烈亚斯创办了几家比特币企业，并推出了一些开源社区项目。他同时是几家比特币和数字货币公司的顾问。

身为比特币方面的畅销书作家，安德烈亚斯发表了大量的文章和博客，也是流行播客“我们来聊比特币（Let’s Talk Bitcoin）”的永久主持人，同时也是全球技术和安全会议的演讲常客。

结语

出现在本书封面上的动物是切叶蚁（学名*Atta colombica*）。切叶蚁，不是一个常见的名字，它是一种生长在热带地区、靠真菌长大的蚂蚁，主要分布在中南美洲，墨西哥及美国南部。除了人类社会，切叶蚁构成了地球上最大、最复杂的动物社会群体。它们因咀嚼树叶的方式而得名，被它们咀嚼过的树叶是它们真菌花园的营养来源。

有翅膀的切叶蚁，无论雌雄，都要集体离开它们的巢穴，参与一场飞行婚礼。雌蚁会与不同的雄蚁交配，收集够300万个精子以形成它新的殖民群落。雌蚁还会在其口下囊存储一些真菌的菌丝，将这些菌丝用来培植它自己的真菌花园。一旦落地，雌蚁的翅膀就会脱落，并开始创建自己的地下殖民地。这些新蚁后的成功率并不高，只有约2.5%能够成功建立一个长寿的殖民地。

当殖民地成熟时，蚂蚁们将根据体型分为不同的种群，每个种群负责不同的功能。通常有四类种群：最小的，最小的工蚁负责照顾年轻蚂蚁及真菌花园；次小的，比最小的稍大，是殖民地的第一道防线，负责巡逻领地及攻击敌人；中等的，是觅食者，它们负责切割树叶，并把树叶碎片搬回巢中；最大的，是工蚁中的战士，负责抵御入侵者，捍卫巢穴安全。最近的研究表明，最大的工蚁也会负责清理觅食通道，并把大件物品搬回巢穴。

很多出现在O’Reilly书籍封面的动物都濒临灭绝，它们对世界来说都非常重要。如果想要了解你如何才能为保护动物提供帮助，请访问animals.oreilly.com。

封面图片来自《海外昆虫》（*Insects Abroad*）。

关于译校者

蔡长春

兴业银行信息科技部架构师，先后服务于证券公司、保险公司、银行等金融机构，长期从事金融系统开发及架构设计工作。对金融科技抱有极大热情，是新技术积极的倡导者和推动者。从2015年至今，潜心研究区块链，对区块链技术在金融行业的应用有深入的研究和丰富的实战经验。

王志涵

光大中兴集团董事长、新加坡量子数字基金创始合伙人、清华大学五道口金融学院全球金融博士、长江商学院EMBA、全球区块链项目资深投资人、知名数字资产投资和管理人。

王勇

博士、国家千人计划专家、光大证券首席风险官，兼任中国证券业协会风险管理专业委员会副主任委员、中国人民大学博士生导师。加拿大达尔豪斯大学数学博士，曾任加拿大皇家银行风险管理部董事总经理。特许金融分析师，并持有FRM（金融风险管理）证书。

对区块链、金融科技、投资、金融衍生产品、风险管理等领域有很深入的研究，著有《金融风险管理》，译有《期权、期货及其他衍生产品》《风险管理与金融机构》《区块链：技术驱动金融》等9部著作。

王立荣

现任山西信托副总裁（级）、汇丰晋信基金管理公司副总经理，兼任中国资产证券化研究院副院长，国家发改委PPP专家库首批入库专家。

16年金融市场从业经历，7年金融机构高管人员任职经历，长期从事股票、债券、利率和股指期货的投资、交易和研究工作，对宏观经济、货币与财政政策、资产证券化与区块链技术等领域有深入的研究和实践经验的积累。